Java Tutorial For Beginners

Welcome to this book on **"Learning Java In 150 Steps"**.

I am **Ranga Karanam**, and I have more than two decades of programming experience.

I love Programming. One of the aims I had when I started `in28minutes` was to make learning programming easy. Thanks for helping us provide amazing courses to 300,000 learners across the world.

At **In28Minutes**, we ask ourselves one question every day: "How do we create awesome learning experiences?"

In this book, you will learn to write **object oriented** code with Java. You will be exposed to a lot of examples, exercises and tips. We will take up a lot of examples, and try and see how to write code for those in Java.

Table of Contents

Our Approach

We did a study on why students give up on programming?

The popular answer was

> Difficulty in writing their first program

Put yourselves in the shoes of a beginner and look at this typical `Java Hello World Example`.

```
package com.in28minutes.firstjavaproject;
public class HelloWorld
{
     public static void main(String[] args) {
            System.out.println("Hello World");
```

```
    }
}
```

A `Programming Beginner` will be overwhelmed by this. I remember how I felt when I saw this almost 20 years back. Stunned.

Why?

- There are a number of keywords and concepts - package, public, class, static, void, String[] and a lot more..
- What if the programmer makes a typo? Will he be able to fix it?

We believe that there has to be a better way to learn programming.

- Why don't we learn programming step by step?
- Why should it not be a lot of fun?
- Why don't we solve a lot of problems and learn programming as a result?

This is the approach we took to writing this guide and develop our introductory programming courses for Java and Python.

> Do you know? The first 3 hours of our Java Course is available here.

Introduction to Programming with Print-Multiplication-Table

Step 01: First Challenge : The Print-Multiplication-Table (*PMT-Challenge*)

Learning to program is a lot like learning to ride a bicycle. The first few steps are the most challenging ones.

Once you use this stepwise approach to solve a few problems, it becomes a habit.

In this book, we will introduce you to Java programming by taking on a few simple problems to start off.

> Having fun along the way is what we will aim to do.

Are you all geared up, to take on your first programming challenge? ***Yes****? Let's get started then!*

Our first *programming challenge* aims to do, what every kid does in math class: reading out a multiplication table.

The *PMT-Challenge*

1. Compute the multiplication table for `5`, with entries from `1` to `10`.
2. Display this table.

```
5 * 1  =  5
5 * 2  = 10
5 * 3  = 15
5 * 4  = 20
5 * 5  = 25
5 * 6  = 30
5 * 7  = 35
5 * 8  = 40
5 * 9  = 45
5 * 10 = 50
```

As part of solving the multiplication table problem, you will be introduced to:

- **JShell**
- **Statements**
- **Expressions**

- **Variables**
- **Literals**
- **Conditionals**
- **Loops**
- **Methods**

Summary

In this step, we:

- Stated our first programming challenge, *PMT-Challenge*
- Identified basic Java concepts to learn, to solve this challenge

Step 02: Introducing `JShell`

`JShell` is a programming tool, introduced in Java SE 9. JShell is a ***REPL*** interface. The term **REPL** refers to this:

- ***'R'*** stands for **R**ead; *Read* the input Java code
- ***'E'*** means **E**val; *Evaluate* the source code
- ***'P'*** translates to **P**rint; *Print* out the result
- ***'L'*** indicates **L**oop; *Loop* around, and wait for the next input

How about starting off exploring Java? Are you game?

Snippet-1: Check the Java programming environment

You can use https://tryjshell.org/ to run the code for the first 25 steps. Or you can Install Java 12+. Here's the troubleshooting section if you face problems.

Launch up command prompt or Terminal.

Let type in `java -version` on the terminal and press enter.

```
in28minutes$>java -version
java version "x.0.1"
Java(TM) SE Runtime Environment (build x.0.1+11)
Java HotSpot(TM) 64-bit Server VM (build x.0.1+11, mixed mode)
in28minutes$>
```

A successful execution displays the version of Java installed your system. You need to have atleast Java 9 to pursue this book.

Snippet-2: Launch JShell

You can launch JShell by typing `jshell` at your terminal.

```
in28minutes$>jshell

|  Welcome to JShell version x.0.1
 |  For an introduction type: /help intro
jshell>
```

When run, this command displays basic information about the installed `JShell` program. A `jshell` prompt then appears, which waits for your input.

Snippet-3: Sample JShell input, using a built-in command

The `JShell` command `/help`, with a parameter `intro`, gives you basic guidelines on how you use the tool.

```
jshell> /help intro

|
| intro
| The jshell tool allows you to execute Java code, getting immediate results.
| You can enter a Java definition (variable, method, class, etc), like: int x =8
| or a Java expression, like: x + x
| or a Java statement or import.
| These little chunks of Java code are called 'snippets'.
| There are also jshell commands that allow you to understand and
| control what you are doing, like: /list
|
| For a list of commands: /help

jshell>
```

Snippet-4: Evaluating an expression

Type `3+4` at the Jshell prompt

```
jshell> 3 + 4
$1 ==> 7
jshell>
```

This was your first real **REPL** cycle! When you type in `3 + 4`, JShell evaluates it and prints the result.

> The entity `$1` is actually a variable name assigned to the result. We will talk about this later.

Snippet-5: Getting out of JShell

The `/exit` command terminates the `JShell` program, and we are back to the terminal prompt.

```
jshell> /exit
|  Goodbye

in28minutes$>
```

Snippet-6: Again, enter JShell and Exit it!

You can now effortlessly launch, feed code to, and exit from `JShell`!

```
in28minutes$> jshell

|  Welcome to JShell version 9.0.1
|  For an introduction type: /help intro
jshell> /exit
|  Goodbye

in28minutes$>
```

Summary

In this step, we learned:

- How to launch `JShell` from our terminal, and run a few commands on it
- How to run Java code on the `JShell` prompt

Step 03: Welcome to Problem Solving

Lets try to break down the *PMT-Challenge* problem to be able to solve it.

```
5 * 1  =  5
5 * 2  = 10
5 * 3  = 15
5 * 4  = 20
5 * 5  = 25
5 * 6  = 30
5 * 7  = 35
5 * 8  = 40
5 * 9  = 45
5 * 10 = 50
```

Here is how our draft steps look like

- Calculate `5 * 3` and print result as `15`
- Print `5 * 3 = 15` (`15` is result of previous calculation)
- Do this ten times, once for each table entry (going from `1` to `10`)

Summary

In this step, we:

- Broke down the *PMT-Challenge* problem into sub-problems

Step 04: Introducing Expressions

The first part of solving our *PMT-Challenge* is to calculate the product of `5` and another number, say `3` .

Let's start up jshell and type `5 X 3` .

```
    in28minutes$> jshell

    |  Welcome to JShell version x.0.1
    |  For an introduction type: /help intro
    jshell> 5 X 3
    | Error:
    | ';' expected
    | 5 X 3
    |  ^
    | Error:
    | not a statement
    | 5 X 3
    |   ^
    | Error:
    | ';' expected
    | 5 X 3
    |    ^
    | Error:
    | missing return statement
    | 5 X 3
    | ^---^
```

You probably look at the symbol 'X' as a multiplier, remembering your school days.

Java does not identify ' `x` ' as the multiplication operator! Java supports multiplication, but only if you use its *predefined* operator, `*` .

Let's type in code shown below:

```
jshell> 5 * 3
$1 ==> 15
jshell>
```

Success!

Let's look at some terminology:

- `5 * 3` is an expression.
- `5` and `3` are operands. They are also called **literals** or constant values.
- `*` is an operator.

Java also has built-in operators to perform other numerical tasks, such as:

- Addition: `+`
- Subtraction: `-`
- Division: `/`
- Modulo arithmetic: `%`

The following examples illustrate how to use them.

```
jshell> 5 * 10
$2 ==> 50
jshell> 5 + 10
$3 ==> 15
jshell> 5 - 10
$4 ==> -5
jshell> 10 / 2
$5 ==> 5
jshell>
```

Your school math memories are still fresh, and the operators don't seem to disappoint either! `+` , `-` and `/` are your bread-and-butter operators.

`%` is the modulo operator, which gives you the remainder when integer division is performed.

```
jshell> 9 % 2
$6 ==> 1
jshell> 8 % 2
$7 ==> 0
jshell>
```

Snippet: Complex Expressions, Anyone?

Java allows you to use more than one operator within an expression.

Let's look at some simple expressions with multiple operators.

```
jshell> 5 + 5 + 5
$8 ==> 15
jshell> 5 + 10 - 15
```

```
$9 ==> 0
jshell> 5 * 5 + 5
$10 ==> 30
jshell> 5 * 15 / 3
$11 ==> 25
```

Each of above expressions have two operators and three operands.

Summary

In this step, we:

- Learned how to construct numeric expressions
- Discovered that operators are predefined symbols
- Combined several operators to form larger expressions

Step 05: Programming Exercise PE-1 (With Solutions)

At this stage, your smile tells us that you enjoy evaluating Java expressions. What if we tickle your mind a bit, to make sure it hasn't fallen asleep?

Okay, here comes a couple of programming exercises.

1. Write an expression to calculate the number of minutes in a day.
2. Write an expression to calculate the number of seconds in a day.

Solution 1

60 (minutes in an hour) multipled by 24 (hours in a day)

```
jshell> 60 * 24
$1 ==> 1440
```

Solution 2

60 (seconds in a minute) multipled by 60 (minutes in an hour) multipled by 24 (hours in a day)

```
$jshell>60 * 60 * 24

$1 ==> 86400
```

Step 06: Operators

Lets look at a few more examples to understand operators.

Invalid Operators

Let's type in `5 ** 6` followed by `5 $ 6` in JShell

```
jshell> 5 ** 6
| Error:
| Illegal start of expression
| 5 ** 6
|     ^

jshell> 5 $ 6
| Error:
| ';' expected
| 5 $ 6
|   ^
| Error:
| not a statement
| 5 $ 6
|    ^
| Error:
| ';' expected
| 5 $ 6
|     ^

jshell> 5 */ 6
| Error:
| Illegal start of expression
| 5 */ 6
|      ^
```

`JShell` was not impressed with our efforts at all!

Java has a set of grammar rules, called its **syntax**. Code that does not follow these rules throw errors. The compiler informs you by listing the errors, and also provides hints on how you can correct them.

Now, why is an error being thrown?

Operators in Java are all predefined, and limited in number. `*` is a valid operator, whereas `**` and `$` were rejected, with error messages.

Understanding Result of Expression with Mixed Types

Let's look at another example:

```
jshell> 5 / 2
$1 ==> 2
jshell>
```

Surprise, Surprise! `JShell` seems to evaluate `5 / 2` to `2` instead of `2.5`. Where did we go wrong?

Read what follows, with the biggest magnifying lens you can find:

The result of an expression when evaluated, depends on the operator context. This context is determined by the operands passed to it

There are two kinds of numbers typically used in programming : integer (1,2,3,...) and floating-point (1.1,2.3, 56.7889 etc). These values are represented by different **types** in Java. Integers are commonly of type `int`, and the floating-point numbers are `double` by default.

In the expression `5/2`, both `5` and `2` are of type `int`. So, the result is also of type `int`.

Let's try with a few floating point numbers:

```
jshell> 5.0 / 2.0
$2 ==> 2.5
```

Both `5.0` and `2.0` are of type `double`, the result is `double`. We get a result of `2.5`, as expected.

Let's do a mixed operation - using a floating point number and integer.

```
jshell> 5.0 / 2
$3 ==> 2.5
```

```
jshell>
```

Among the types `int` and `double`, `double` is considered to be a *wider type*. When you perform a numeric operation between two types, the result will be of the wider type.

Understanding Precedence of Operators

Let's look at few complex examples of expressions with more than one operator.

```
jshell> 5 + 5 * 6
$1 ==> 35
jshell> 5 - 2 * 2
$2 ==> 1
jshell> 5 - 2 / 2
$3 ==> 4
jshell>
```

Surprised with the results? You might expect 5 + 5 * 6 evaluate to 10 * 6 i.e. 60. Howeever, we got `35`!

> We write English left-to-right, and carry this habit to calculations as well.

*In expressions with multiple operators, the order of sub-expression evaluation depends on **operator precedence**.*

The basic rules for operator precedence are actually quite simple (we will look at other rules a little later).

The operators in the set { `*`, `/`, `%` } have higher precedence than the operators in the set { `+`, `-` }.

In the expression `5 + 5 * 6` : 5*6 is evaluated first. So, 5 + 5 * 6 becomes 5 + 30 i.e. 35.

`5 - 2 * 2` and `5 - 2 / 2` are evaluated by following the same rules.

Use paranthesis for clear code

Java provides syntax elements, called **parentheses** `(` and `)`, to group parts of an expression.

```
jshell> (5 - 2) * 2
$4 ==> 6
jshell> 5 - (2 * 2)
$5 ==> 1
jshell>
```

When you put an expression in parenthesis, it is evaluated first. (5 - 2) * 2 => 3 * 2 => 6.

Parentheses lead to better readability as they reduce confusion and help avoid errors.

The old adage ***A stitch in time saves nine*** rings very true here. Use parentheses to make your code easy to read, even when they might not be necessary.

Summary

In this step, we:

- Discovered that operators are all predefined
- Learned that result of operation depends on operand types
- Understood what operator precedence means
- Used parentheses to group parts of an expression

Step 07: Introducing Console Output

We have computed the product of two literals (as in `5 * 3`) in earlier steps.

The next step is to print this result in a customized format - `5 * 3 = 15`.

How do we do this?

Let try typing it in into JShell

```
jshell> 5 * 3 = 15
|  Error:
|  unexpected type
|    required: variable
|    found:    value
|  5 * 3 = 15
|  ^___^
```

Hmm! Error.

How do we print text?

Java has a built-in utility method called `System.out.println()`, that displays text on the console.

```
jshell> System.out.println(3*4)
12
```

We formed an expression, `3*4`, and *passed* it to `System.out.println()`, which is a built-in Java **method**.

`System.out.println(3*4)` is an example of a **statement**. It is a **method call**.

The syntax rules for method calls are quite strict, and all its parts are mandatory.

```
jshell> System.out.println3*4)
| Error:
| ';' expected
| System.out.println3*4)
|___________________^
| Error:
| cannot find symbol
|       symbol: variable println3
| System.out.println3*4)
| ^______________________^

jshell> System.out.println 3*4
|  Error:
|  ';' expected
|  System.out.println 3*4
|                    ^
|  Error:
|  cannot find symbol
|    symbol:   variable println
|  System.out.println 3*4
|  ^_________________^
```

What if we want to print an entry of the Multiplication Table, as part of our solution to *PMT-Challenge*? In other words, how do we print the exact text `5 * 2 = 10` on the console?

```
jshell> System.out.println(5 * 2 = 10)
| Error:
| unexpected type
| required:  variable
| found:     value
| System.out.println(5 * 2 = 10)
|____________________^________^
```

You wanted to print `5 * 2 = 10` on the console. However, we see that we cannot pass `5 * 2 = 10` as an argument to `System.out.println()`.

`5 * 2 = 10` is not a single value. It is a piece of text with numeric characters, `*` and `=`.

In Java, we represent text using `String`. A `String literal` is a sequence of characters, enclosed within **double quotes**: `"` and `"`.

```
jshell> System.out.println("5 * 2 = 10")
| 5 * 2 = 10
```

Congratulations! You have now figured out how to display not just numbers on the console, but text as well!

Summary

In this step, we:

- Were introduced to the `System.out.println()` method for console output
- Used this utility to print a single *PMT-Challenge* table entry

Step 08: Programming Exercise PE-02

Try and solve the following exercises:

1. Print `Hello World` onto the console.
2. Print `5 * 3`, as is.
3. Print the calculated value of `5 * 3`.
4. Print the number of seconds in a day, using the `System.out.println` method.
5. Do a syntax revision for the code that you write for each of the above exercises. In your code, identify the following elements:
   - Numeric and string literals
   - Expressions
   - Operators
   - Operands
   - Method calls

Step 09: Solutions to PE-02

Solution 1

```
jshell> System.out.println("Hello World")
Hello World
```

Solution 2

```
jshell> System.out.println("5 * 3")
5 * 3
jshell>
```

Solution 3

```
jshell> System.out.println(5 * 3)
15
```

Solution 4

```
jshell> System.out.println(60 * 60 * 24)
86400
```

Step 10: Whitespace, Case sensitiveness and Escape Characters

The term *whitespace* refers to any sequence of continuous space, tab or newline characters.

Whitespace

Let's see a few examples of whitespace in action.

Whitespace affects the output when used in-and-around string literals.

```
jshell> System.out.println("Hello World")
Hello World
jshell> System.out.println("Hello      World")
Hello      World
jshell> System.out.println("HelloWorld")
HelloWorld
```

Whitespace is ignored by the compiler when used around numeric expressions.

```
jshell> System.out.println(24 * 60 * 60)
86400
jshell> System.out.println(24    *    60    *    60)
86400
jshell>
```

Case Sensitive

Java is case sensitive.

`System.out.println()` involve pre-defined Java elements : the `System` **class** name, the `out` **variable** name,and the `println` **method** name. All are *case-sensitive*. If any character in these names is used with a different case, you get an error.

```
jshell> system.out.println("Hello World")
| Error:
| package system does not exist
| system.out.println("Hello World")
| ^-------^

jshell> System.Out.println("Hello World")
| Error:
| cannot find symbol
| symbol: variable Out
```

```
    | System.Out.println("Hello World")
    | ^-------------^

    jshell> System.out.Println("Hello World")
    | Error:
    | cannot find symbol
    | symbol: method Println(java.lang.string)
    | System.out.Println("Hello World")
    | ^---------------------^

    jshell>
```

Inside a string literal, the case of characters do not cause errors. The literal will be taken in and printed, as-is.

```
jshell> System.out.println("hello world")
hello world
jshell> System.out.println("HELLO WORLD")
HELLO WORLD
```

Escape Characters

An **escape character** is a special symbol, used with another regular symbol to form an **escape sequence**. In Java, the ' \ ' (*back-slash*) character plays the role of an escape character. This escape sequence changes the original usage of the symbols.

If you want to print the string **delimiter**, the `"` character, you need to escape it with a `\`. Without it, a `"` character within a string literal causes an error!

```
    jshell> System.out.println("Hello "World")
    | Error:
    | ')' expected
    | System.out.println("Hello "World")
    |                            ^

    jshell> System.out.println("Hello \"World")
    Hello "World
    jshell>
```

The escape sequence `\n` inserts a *newline*.

```
jshell> System. out.println("Hello \n World")
Hello
 World
jshell> System.out.println("Hello n World")
Hello n World
jshell> System.out.println("Hello\nWorld")
Hello
World
jshell>
```

The escape sequence `\t` inserts a *tab*.

```
    jshell> System.out.println("Hello \t World")
    Hello    World
    jshell> System.out.println("Hello t World")
    Hello t World
    jshell> System.out.println("Hello\tWorld")
```

```
Hello    World
jshell>
```

How do you print a `\` ?

```
jshell> System.out.println("Hello \ World")
|  Error:
|  illegal escape character
|  System.out.println("Hello \ World")
```

You would need to escape it with another `\` . Printing `\\` outputs the symbol `\` to the console.

```
jshell> System.out.println("Hello \\ World")
Hello \ World
jshell> System.out.println("Hello \\\\ World")
Hello \\ World
```

Summary

In this step, we:

- Were introduced to method call syntax, with `System.out.println()`
- Discovered the uses of whitespace characters
- Learned about Java escape sequences

Step 11: More On Method Calls

Let's look at method calls with a few more examples.

You know how to invoke a method with a single argument, courtesy `System.out.println(3*4)` . Other scenarios do exist, such as

- Calling a method without any arguments
- Calling a method with several arguments

Let's check them out, using the built-in methods in Java `Math` class.

Method without parameters

In method calls, parentheses are a necessary part of the syntax. Don't leave them out!

`Math.random()` prints a random real number in the range `[0 .. 1]` , a different one on each call

```
jshell> Math.random
| Error:
| cannot find symbol
|    symbol: Math.random
| Math.random
| ^-------------- ^
jshell> Math.random()
$1 ==> 0.0424279106074651_
jshell> Math.random()
$2 ==> 0.8696879746593543
jshell> Math.random()
$3 ==> 0.8913591586787125
```

Method with multiple parameters

How do we call `Math.min` with two parameters `23` and `45` ?

```
jshell> Math.min 23 45
| Error
| cannot find symbol
| symbol: variable min
| Math.min 23 45
| ^---------^
jshell> Math.min(23 45)
| Error
| ')' expected
| Math.min 23 45
| ---------------^
```

While making method calls, the programmer must

- Enclose all the parameters within parentheses
- Separate individual parameters within the list, using the comma symbol ' , '.

```
jshell> Math.min(23, 45)
$4 ==> 23
jshell> Math.min(23, 2)
$5 ==> 2
jshell> Math.max(23, 45)
$6 ==> 45
jshell> Math.max(23, 2)
$7 ==> 2
jshell>
```

`Math.min()` returns the minimum of two given numbers. `Math.max()` returns the maximum of two given numbers.

Summary

In this step, we:

- Understood how zero, or multiple parameters are passed during a method call

Step 12: More Formatted Output

`System.out.println()` can accept one value as an argument at a maximum.

To display the multiplication table for `5` with a calculated value, we need a way to print both numbers and strings.

For this we would need to use another built-in method `System.out.printf()` .

When `System.out.printf()` is called with a *single* string argument, it prints some illegible information. For now, it suffices to know, that this information is about the built-in type `java.io.PrintStream` .

```
jshell> System.out.printf("5 * 2 = 10")
5 * 2 = 10$1 ==> java.io.PrintStream@4e1d422d
jshell>
```

The good news is, that if we call the `println()` method on this, the illegible stuff disappears.

```
jshell> System.out.printf("5 * 2 = 10").println()
5 * 2 = 10
```

The method `System.out.printf()` accepts a variable number of arguments:

- The first argument specifies the print format. This is a string literal, having zero or more **format specifiers**. A format specifier is a predefined literal (such as `%d`), that formats data of a particular type (`%d` formats data of type `int`).
- The trailing arguments are expressions,

Lots of theory? Let's break it down. Let's look at an example.

```
jshell> System.out.printf("5 * 2 = %d", 5*2).println()
5 * 2 = 10
jshell>
```

`System.out.printf("5 * 2 = %d", 5*2).println()` gives an output `5 * 2 = 10`. `%d` is replaced by the calculated value of `5*2`.

Let's play a little more with `printf`:

```
jshell> System.out.printf("%d %d %d", 5, 7, 5).println()
5 7 5
```

Let's try to display a calculated value. In the example below `5*7` is calculated as `35`.

```
jshell> System.out.printf("%d %d %d", 5, 7, 5*7).println()
5 7 35
```

Let's use this to print the output in the format that we want to use for multiplication table:

```
jshell> System.out.printf("%d * %d = %d", 5, 7, 5*7).println()
5 * 7 = 35
```

Congratulations. We are able to calculate `5*7` and print `5 * 7 = 35` successfully.

Exercise

1. Print the following output on the console: `5 + 6 + 7 = 18`. Use three literals `5`, `6`, `7`. Calculate 18 as `5 + 6 + 7`.

Solution

```
jshell> System.out.printf("%d + %d + %d = %d", 5, 6, 7, 5 + 6 + 7).println()
5 + 6 + 7 = 18
jshell>
```

Playing with `System.out.printf()`

In the example below, there are four format specifiers (`%d`) and only three value arguments `5, 6, 7`.

```
jshell> System.out.printf("%d + %d + %d = %d", 5, 6, 7).println()
5 + 6 + 7 = | java.util.MissingFormatArgumentException thrown: Format specifier '%d'
| at Formatter.format (Formatter.java:2580)
| at PrintStream.format (PrintStream.java:974)
| at PrintStream.printf (PrintStream.java:870)
| at (#52:1)
```

In a call to `System.out.printf()` , if the number of format specifiers exceeds the number of trailing arguments, the Java run-time throws an *exception*.

If the number of format specifiers is less than the number of trailing arguments, the compiler simply ignores the excess ones.

```
jshell> System.out.printf("%d + %d + %d", 5, 6, 7, 8).println()
5 + 6 + 7
jshell>
```

More Format Specifiers

String values can be printed in `System.out.printf()` , using the format specifier `%s` .

```
jshell> System.out.printf("Print %s", "Testing").println()
Print Testing
```

Earlier, we used %d to print an `int` value. You cannot use %d to display floating point values.

```
jshell> System.out.printf("%d + %d + %d", 5.5, 6.5, 7.5).println()
| java.util.IllegalFormatConversionException thrown: d != java.lang.Double
| at Formatter$FormatSpecifier.failedConversion(Formatter.java:4331)
| at Formatter$FormatSpecifier.printInteger(Formatter.java:2846)
| at Formatter$FormatSpecifier.print(Formatter.java:2800)
| at Formatter.format(Formatter.java:2581)
| at PrintStream.format(PrintStream.java:974)
| at PrintStream.print(PrintStream.java:870)
| at #(57:1)
```

Floating point literals (or expressions) can be formatted using `%f` .

```
jshell> System.out.printf("%f + %f + %f", 5.5, 6.5, 7.5).println()
5.500000 + 6.500000 + 7.500000
jshell>
```

Summary

In this step, we:

- Discovered how to do formatted output, using `System.out.printf()`
- Stressed on the number and sequence of arguments for formatting
- Explored the built-in format specifiers for primitive types

Step 13: Introducing Variables

In the previous steps, we worked out how to print a calculated value as part of our multiplication table.

```
jshell> System.out.printf("%d * %d = %d", 5, 1, 5 * 1).println()
5 * 1 = 5
```

How do we print the entire multiplication table?

We can do something like this.

```
jshell> System.out.printf("%d * %d = %d", 5, 1, 5 * 1).println()
5 * 1 = 5
jshell> System.out.printf("%d * %d = %d", 5, 2, 5 * 2).println()
5 * 2 = 10
jshell> System.out.printf("%d * %d = %d", 5, 3, 5 * 3).println()
5 * 3 = 15
jshell> System.out.printf("%d * %d = %d", 5, 4, 5 * 4).println()
5 * 4 = 20
jshell>
```

Too much work. Isn't it?

If you carefully observe the code, these statements are very similar.

What's changing? The number in the third and fourth parameter slots changes from 1 to 4.

Wouldn't it be great to have something to represent the changing value?

Welcome variables.

```
jshell> int number = 10
number ==> 10
jshell>
```

Terminology and Background

In the statement `int number = 10`,

- `number` is a **variable**.
- The literal `number` is the variable **name**.
- The Java *keyword* `int` specifies the **type** of `number`.
- The literal `10` provided `number` with an **initial value**.
- This entire statement is a **variable definition**.

The effects of this variable definition are:

- A specific location in the computer's memory is reserved for `number`.
- This location can now hold data of type `int`.
- The value `10` is stored here.

You can change the value of number variable:

```
jshell> number = 11
number ==> 11
```

Above statement is called variable **assignment**.

An assignment causes the value stored in the memory location to change. In this case, `10` is replaced with the value `11`.

You can look up the value of number variable.

```
jshell> number
number ==> 11
```

You can change the value of a variable multiple times.

```
jshell> number = 12
number ==> 12
jshell> number
number ==> 12
```

Let's create another variable:

```
jshell> int number2 = 100
number2 ==> 100
jshell> number2 = 101
number2 ==> 101
jshell> number2
number2 ==> 101
jshell>
```

The statement `int number2 = 100` defines a *distinct* variable `number2`.

How do we use variables to simplify and improve the solution to PMT-Challenge?

Let's take a look.

```
jshell> System.out.printf("%d * %d = %d", 5, 4, 5*4).println()
5 * 4 = 20
```

Let's define a variable `i`, and initialize it to `1`.

```
jshell>int i = 1
i ==> 1
jshell> i
i ==> 1
jshell> 5*i
$1 ==> 5
```

Let's update the multiplication table printf to use the variable i.

```
jshell> System.out.printf("%d * %d = %d", 5, i, 5*i).println()
5 * 1 = 5
```

How do we print `5 * 2 = 10`?

We update `i` to `2` and execute the same code as before.

```
jshell> i = 2
i ==> 2
jshell> 5*i
$2 ==> 10
jshell> System.out.printf("%d * %d = %d", 5, i, 5*i).println()
5 * 2 = 10
jshell>
```

You can update the value of `i` to any number.

The previous statement would print the corresponding multiple with 5.

```
jshell> i = 3
i ==> 3
jshell> System.out.printf("%d * %d = %d", 5, i, 5*i).println()
5 * 3 = 15
jshell> i = 10
i ==> 10_
jshell> System.out.printf("%d * %d = %d", 5, i, 5*i).println()
5 * 10 = 50
jshell>
```

By varying the value of `i`, we are able to print different multiples of 5 with the same statement.

Congratulations! You made a major discovery. Variables.

Summary

In this step, we:

- Understood the need for variables
- Observed what different parts of a variable definition mean
- Seen what goes on behind-the-scenes when a variable is defined

Step 14: Programming Exercise PE-03 (With solution)

1. Create three integer variables `a`, `b` and `c`.
    - Write a statement to print the sum of these three variables.
    - Change the value of `a`, and then print this sum.
    - Then again, change the value of `b` and print this sum.

Solution to PE-03

```
jshell>int a = 5
a ==> 5
jshell>int b = 7
b ==> 7
jshell>int c = 11
c ==> 11
jshell>System.out.printf("a + b + c = %d", a+b+c).println()
a + b + c = 23
jshell>a = 2
a ==> 2
jshell>System.out.printf("a + b + c = %d", a+b+c).println()
a + b + c = 20
jshell>b = 9
b ==> 9
jshell>System.out.printf("a + b + c = %d", a+b+c).println()
a + b + c = 22
jshell>
```

Step 15: Using Variables

Variables should be declared before use.

```
jshell>newVariable
| Error:
| cannot find symbol
| symbol: newVariable
| newVariable
  ^_____________^
```

Java is a **strongly typed** programming language. This means two things:

- Every variable in a program must be declared, with a type.
- Values assigned to a variable should be:
  - same type as the variable's type, or
  - **compatible** with the variable's type

```
jshell> int number = 5.5
| Error:
| incompatible types: possible lossy conversion from double to int
| int number = 5.5
|_____________^___^
jshell>
```

The variable `number` is an integer, mathematically a number. The constant `5.5` is a number as well.

Why does it result in error?

`5.5` is a floating-point number of type `double`. We are trying to store a `double` inside a memory slot meant for `int`.

Let's consider another example:

```
jshell> int number = "Hello World"
| Error:
| incompatible types: java.lang.String cannot be converted to int
| int number = "Hello World"
|_____________^_______________^
jshell>
```

`number` is an `int` variable, and we are trying to store a `String` value `"Hello World"`. ***Not allowed.***

Summary

In this step, we:

- Observed how declaring a variable is important
- Understood the compatibility rules for basic Java types

Step 16: Variables: Behind-The-Scenes

Giving a value to a variable, during its declaration, is called its initialization.

```
jshell> int number = 5
number ==> 5
jshell>
```

The statement `int number = 5` combines a declaration and the initialization of `number`.

The next statement `int number2 = number` is a little different.

```
jshell> int number2 = number
```

```
number2 ==> 5
jshell> number2
number2 ==> 5
jshell>
```

The initial value for `number2` is another variable, which is previously defined (`number`).

Here's what goes on behind the scenes with `int number2 = number` :

- A memory slot is allocated for `number2` with size of `int` .
- Value stored in `number` 's slot is copied to `number2` 's slot.

In example below, `a = 100` , `c = a + b` , `a = c` are called `assignments` .

```
jshell> int a = 5
a ==> 5
jshell> int b = 10
b ==> 10
jshell> a = 100
a ==> 100
jshell> int c
c ==> 0
jshell> c = a + b
c ==> 110
jshell> a = c
a ==> 110
jshell> int d = b + c
d ==> 120
jshell>
```

Variable **assignment** is allowed to happen in multiple ways:

- From a literal value, to a variable, having compatible types. The statement `a = 100` is one such example.
- From a variable to another variable, of compatible types.The statement `a = c` illustrates this case.
- From a variable expression to a variable. This expression can be formed with variables and literals of compatible types, and is evaluated before the assignment. The statement `c = a + b` below is an example of this.

An assignment to a constant literal is **not allowed**.

```
jshell> 20 = var
| Error:
| unexpected type
| required : variable
| found : value
| 20 = var
| ^^

jshell>
```

Summary

In this step, we discussed how to provide variables with initial values and the basics of assigment.

Step 17: Naming Variables

The syntax rules for variable definition are quite strict. Do we have any freedom with the way we name variables? The answer is "yes", to some extent.

Variable Name Rules

Here are the Java rules for variable names, in a nutshell:

A variable name can be a sequence, in any order, of

- Letters [`A-Z, a-z`]
- Numerical digits [`0-9`]
- The special characters ' `$` ' ("dollar") and ' _ ' ("underscore")

With the following exceptions:

- The name cannot start with a numerical digit [`0-9`]
- The name must not match with a predefined Java **keyword**

Let's now see what kinds of errors the compiler throws if you violate these naming rules.

Error : Using a invalid character **-**

```
jshell> int test-test
| Error:
| ';' expected
| int test-test
|          ^
```

Error : Starting name of a variable with a number

```
jshell> int 2test
| Error:
| '.class' expected
| int 2test
|     ^
| Error:
| not a statement
| int 2test
|      ^__^
| Error:
| unexpected type
| required: value
| found: class
| int 2test
| ^__^
| Error:
| missing return statement
| int 2test
| ^-------^

jshell>
```

Error : Using a keyword as the name of variable

In Java, certain special words are called `keywords` . For example, some of the data types we looked at - `int` , `double` . These `keywords` cannot be used as variable names.

```
jshell> int int
   ...> ;
| Error:
| '.class' expected
| int int
|      ^
| Error:
| unexpected type
| required: value
| found: class
| int int
| ^--^
| Error:
| missing return statement
| int int
| ^------...
```

Variable Naming Conventions

Good programmers write readable code. Giving proper names to your variables makes your code easy to read.

In a football scorecard application, using a name `s` for a score variable is vague. Something like `score` carries more meaning, and is preferred.

```
jshell> int s
s ==> 0
jshell> int score
score ==> 0
jshell>
```

> The Java community, with its wealth of experience, *suggests* that programmers follow some **naming conventions**. Above examples is one of these. Violation of these rules does not result in compilation errors. That's why, these are called **conventions**.

In Java, another convention is to use **CamelCase** when we have multiple words in variable name.

```
jshell> int noOfGoals
noOfGoals ==> 0
```

`noOfGoals` is easier to read than `noofgoals` .

Both `noOfGoals` and `NoOfGoals` are equally readable.

But in Java, the convention followed is to start variable names with a lower case letter. So, for a variable, `noOfGoals` is preferred.

*** NOT RECOMMENDED***

```
jshell> int NoOfGoals
NoOfGoals ==> 0
```

Your variable names can be very long.

```
jshell> int iThinkThisIsQuiteALongName
iThinkThisIsQuiteALongName ==> 0
jshell> int iThinkThisIsSuchALongNameThatIMightNeverUseSuchALongNameAgain
iThinkThisIsSuchALongNameThatIMightNeverUseSuchALongNameAgain ==> 0
```

However, avoid using very long variable names.

Use the shortest meaningful and readable name possible for your variables.

Summary

In this step, we:

- Explored the Java language rules for variable names
- Understood that there are conventions to guide us
- Looked at a popular variable naming conventions

Step 18: Introducing Primitive Types

The table below lists the **primitive types** in Java.

Type of Values	Java Primitive Type	Size (in bits)	Range of Values	Example
Integral Values	byte	8	`-128` to `127`	`byte b = 5;`
Integral Values	short	16	`-32,768` to `32,767`	`short s = 128;`
Integral Values	int	32	`-2,147,483,648` to `2,147,483,647`	`int i = 40000;`
Integral Values	long	64	`-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`	`long l = 2222222222;`
Floating-Point Values	float	32	approximately ±3.40282347E+38F. NOT very precise (avoid for financial/scientific math)	`float f = 4.0f;`
Floating-Point Values	double	64	approximately ±1.79769313486231570E+308. NOT very precise, but better than float (also avoid for financial/scientific math)	`double d = 67.0;`
Character Values	char	16	`'\u0000` to `'\uffff`	`char c = 'A';`
Boolean Values	boolean	1	`true` or `false`	`boolean isTrue = false;`

Let's now look at how we create data of these types, and store them in memory.

We choose type for data, based on:

- The type of data we want to store
- The range of values we would want to store

> Note: In the above examples, we used semi-colon `;` character at the end. This is the Java **statement separator**, which is not mandatory in `JShell` for single statements. However, it is a must when you use *code editors* and *IDE*'s, to write Java code.

Integer Types

The only difference among the integer types is their storage capacity.

```
    jshell> byte b = 5
```

```
b ==> 5
jshell> short s = 128
s ==> 128
jshell> int i = 40000
i ==> 40000
jshell> long l = 2222222222
l ==> 2222222222
jshell>
```

Floating Value Types

`double` is the default type for floating type values with size 64 bits.

```
jshell> double d = 67.0
d ==> 67.0
```

`float` occupies 32 bits. `float` literals need to have a suffix ' `f` ' (as in `4.0f`).

```
jshell> float f = 4.0f
f ==> 4.0
```

If this suffix `f` is omitted, a floating-point number is assumed to be a `double` . You cannot store a 64 bit value into 32 bits.

```
jshell> float f2 = 4.0
| Error:
| incompatible types: possible lossy conversion from double to float
| float f2 = 4.0
|            ^-^
```

You can store `float` value in `double` . Remember, both types store similar kind of values and a `float` value is only 32 bits where as `double` has 64 bits.

```
jshell> double d2 = 67.0f
d2 ==> 67.0
```

Character Type

The character type `char` can store a single character symbol. The symbol must be enclosed within a pair of single quotes, `'` and `'` .

```
jshell> char c = 'A'
c ==> 'A'
```

Following are a few char declaration errors: Not using single quotes and trying to store multiple characters.

```
jshell> char ch = A
| Error:
| cannot find symbol
| symbol: variable A
| char ch = A
|           ^

jshell> char cab = 'AB'
| Error:
| unclosed character literal
| char cab = 'AB'
|            ^
```

Boolean Type

The concept of a `boolean` type is rooted in mathematical logic. The data type can store two possible values, `true` and `false`. Both are case-sensitive labels, but not enclosed in quotes.

```
jshell> boolean isTrue = false
isTrue ==> false
jshell> isTrue = true
isTrue ==> true
jshell> boolean isFalse = True
| Error:
| cannot find symbol
| symbol: variable True
| boolean isFalse = True
|                   ^__^

jshell> isFalse = False
| Error:
| cannot find symbol
| symbol: variable False
| isFalse = False
|           ^___^
```

`boolean` data are mostly used in expressions used to form logical conditions in your code. We will talk more about this - when we explore Java conditionals.

Summary

In this step, we:

- Looked at the primitive data types provided in Java
- Understood the categories into which these types are divided
- Saw what kind of data can be stored in each primitive type

Step 19: Choosing A Data Type

How do we choose the data type for a variable? Let's look at a few examples.

Example 1 : Goals in a Football match

Consider a sports broadcaster that writes a program to track football scores. In a football game there are two teams playing. A game is generally 90 minutes long, with extra time pushing it to 120 minutes at the most. Practically speaking, the scorelines are never huge. From a practical standpoint, the number of goals scored in a match would never exceed 7200 (the number of seconds in 120 minutes). Playing it a bit safe, a `short` data type would be enough to store the score of each side (though a `byte` would be more than enough).

```
jshell> short numTeamAGoals = 0
numTeamAGoals ==> 0
jshell> short numTeamBGoals = 0
numTeamBGoals ==> 0
jshell>
```

Example 2 : How do we store population of the world?

We know that the global count of humans is well over 7 billion now, so the only integer data type that can store it is a `long`.

```
jshell> long globalPopulation = 7500000000;
globalPopulation ==> 7500000000
jshell>
```

Example 3 : How do we store average rainfall in a month?

Rainfall is usually measured in millimeters (*mm*), and we know computing an average over 30 days is very likely to yield a floating-point number. Hence we can use a `float`, or for more accuracy, a `double` to store that average.

```
jshell> float avgJanRainfall = 31.77f;
avgJanRainfall ==> 31.77
jshell> double averageJanRainfall = 31.77
averageJanRainfall ==> 31.77
jshell>
```

Example 4 : Grades of a Student

Classroom grades in high school are generally A, B, C , ...

For a program that generates a grade report, a `char` type would be the best bet.

```
jshell> char gradeA = 'A'
gradeA ==> 'A'
jshell> char gradeB = 'B'
gradeB ==> 'B'
jshell> char gradeC = 'C'
gradeC ==> 'C'
jshell> char gradeD = 'D'
gradeD ==> 'D'
jshell> char gradeF = 'F'
gradeF ==> 'F'
jshell>
```

Example 5 : Is a Number Odd or Even?

A `boolean isNumEven` is a good bet, with a default initial value of `false`. It can be changed to `true` if the number turns out to be even.

```
jshell> boolean isNumEven
isNumEven ==> false
jshell> isNumEven = true
isNumEven ==> true
jshell>
```

Summary

In this step, we:

- Understood how to think while choosing a data type
- Explored cases for integer, floating-point, character and boolean data

Step 20: The Assignment Operator `=`

We were first exposed to the assignment operator while discussing variable assignment.

We are allowed to use expressions to assign values to variables, using code such as `c = a + b`.

Let's look at a few additional ways assignment can be done.

```
jshell> int i = 10
i ==> 10
jshell> int j = 15
j ==> 15
jshell> i = j
i ==> 15
jshell>
```

Let's consider the following example:

```
jshell> i = i * 2
i ==> 30
jshell>
```

The same variable `i` appears on both the right-hand-side (*RHS*) and left-hand-side (*LHS*). How would that make any sense?

The expression on the *RHS* is *independently* evaluated first. The value we get is next copied into the memory slot of the *LHS* variable.

Using the logic above, the assignment `i = i * 2` actually updates the value of `i` (initially `15`) to `30` (doubles it).

Java also does the right thing when it encounters the puzzling statement `i = i + i`, where `i` is all over the place! The code doubles the value of `i`, and this reflects on the `JShell` output.

```
jshell> i = i + i
i ==> 60
```

The next example is self explanatory.

```
jshell> i = i - i
i ==> 0
```

Variable Increment and Decrement

We have already seen increment by assignment:

```
jshell> int i = 0
i ==> 0
jshell> i = i + 1
i ==> 1
jshell> i = i + 1
i ==> 2
jshell> i = i + 1
i ==> 3
jshell>
```

Conversely, the statement `i = i - 1` is called a variable **decrement**. This can also be done repeatedly, and `i` changes value every time.

```
jshell> i = i - 1
i ==> 2
jshell> i = i - 1
i ==> 1
jshell> i = i - 1
i ==> 0
jshell>
```

Summary

In this step, we:

- Looked at the assignment operator `=` in a more formal way
- Looked at heavily used scenarios for assignment, such as increment and decrement

Step 21: Assignment Puzzles, and *PMT-Challenge* revisited

Now that we understand how to define, assign to, and use variables in expressions, it's time to push the boundaries a bit. The following examples explore how we could play around with variables in our code.

Snippet-1 : Pre- and Post- Increment and Decrement

There are two short-hand versions for increment. `number++` denotes *post-increment*. Conversely, `++number` means **pre-increment**.

Operator `++` can be applied only to variables of integer types, which are `byte` , `short` , `int` and `long` .

`number++` is equivalent to the statement `number = number + 1`

```
jshell> int number = 5
number ==> 5
jshell> number++
$1 ==> 5
jshell> number
number ==> 6
jshell> ++number
$2 ==> 7
jshell> number
number ==> 7
jshell>
```

`number--` is *post-decrement*. On the other hand, `--number` is **pre-decrement**.

```
jshell> number--
$3 ==> 7
jshell> number
number ==> 6
jshell> --number
$4 ==> 5
jshell> number
number ==> 5
jshell>
```

What is the difference between prefix and postfix vesions?

Although both prefix and postfix versions achieve the same visible result in above examples, there is a slight difference. We will explore this later.

Snippet-2 : Compound Assignment Operators

The compound assignment operator combines the `=` with a numeric operator.

`i += 2` : Add `2` to the current value of `i`

```
jshell> int i = 1
i ==> 1
jshell> i = i + 2
i ==> 3
jshell> i += 2
$1 ==> 5
jshell>
```

`i -= 1` : Subtract `1` from the current value of `i`

```
jshell> i
i ==> 5
jshell> i -= 1
$2 ==> 4
jshell> i
i ==> 4
jshell>
```

`i *= 4` : Multiply `i` with `4` , and store the result back into `i`

```
jshell> i *= 4
$3 ==> 20
jshell> i
i ==> 20
jshell>
```

`i /= 4` : Divide `i` by `4` , and store the result back into `i`

```
jshell> i /= 4
$4 ==> 5
jshell> i
i ==> 5
jshell>
```

`i %= 2` : Divide `i` by `2` , and store the **remainder** back into `i`

```
jshell> i %= 2
$5 ==> 1
jshell> i
i ==> 1
jshell>
```

Summary

In the step, we:

- Looked at the built-in Java increment and decrement operators
- Explored the side-effects of prefix and postfix versions of these operators
- Seen the advantages of compound assignment

Step 22: Some `JShell` Usage Tips

- Shortcuts
    i. Toggling to limits of current prompt entry
        - '`Ctrl+a`': to start of line
        - '`Ctrl+e`': to end of line
    ii. Moving up and down history of completed prompt inputs
        - *up-arrow* key : move back in time
        - *down-arrow* key : move forward in time
    iii. Reverse-keyword-search
        - Search for a term, in reverse order of JShell prompt inputs history : input '`Ctrl+r`'
        - Scrolling within matches (in reverse order of history) : repeatedly input '`Ctrl+r`'
- `/exit` : resets and exits the JShell session, clearing all stored variables values.
- JShell internal variables : `$1` , `$2` , and so on.
    - When expressions involving literals and/or previously declared variables are entered as-is, without assignment, those expressions are still evaluated. The result is stored in such an internal variable, and is available for use until the JShell session is reset.
- Variable Declaration rules relaxation
    i. Single Java statements need not be terminated with the separator '`;`'.
    ii. If multiple statements are entered on a single line at the prompt, successive statements must be separated by the '`;`'. However, the trailing '`;`' can still be omitted.

Summary

In this step, we:

- Explored some useful keyboard shortcuts for `JShell` commands
- Understood the idea behind `JShell` internal variables
- Observed how `JShell` reduces typing effort, by relaxing some coding rules

Step 23: Introducing Conditionals - the `if`

The *PMT-Challenge* needs us to print a total of 10 table entries. The `for` **loop** is a suitable iteration mechanism to get this done. The word *loop* means exactly what the English dictionary says.

As `i` varies from `1` through `10`, do some stuff involving `i`

For the *PMT-Challenge*, `do some stuff` would be the call to `System.out.printf()`.

The way a `for` loop is structured is:

```
for(initialization; condition; update) {
    statement1;
    statement2;
}
```

Here's how above code runs

1. Execute `initialization` code.
2. If `condition` evaluates to true, execute `statements`. Else, go out of loop.

3. Execute `update`
4. Repeat 2 and 3

Does it sound complex? Let's break it down.

Let's start with understanding what `condition` is all about.

Logical Operators

`condition` represents a logical value(a `true` or a `false`).

Turns out Java has a class of **logical operators**, which can be used with operands within **logical expressions**, evaluating to a `boolean` value.

While in school, you probably used the `=` symbol to compare numbers.

The world of Java is a little different. `=` is assignment operator. `==` is the comparison operator.

```
jshell> int i = 10
i ==> 10
jshell> i == 10
$1 ==> true
jshell> i == 11
$2 ==> false
```

These are other comparison operators as well, such as `<` and `>` .

```
jshell> i < 5
$3 ==> false
jshell> i > 5
$4 ==> true
```

`<=` and `>=` are simple extensions.

```
jshell> i <= 5
$4 ==> false
jshell> i <= 10
$5 ==> true
jshell> i >= 10
$6 ==> true
jshell>
```

Conditionals: The "if" Statement

We would want to execute specific lines of code only when a condition is true.

Enter `if` statement.

An `if` statement is structured this way:

```
if (condition) {
    statement;
}
```

`statement` is executed only if `condition` evaluates to `true` .

Typically, all the code that we write is executed always.

In the example below, `i is less than 5` is always printed.

```
jshell> int i = 10
i ==> 10
jshell> System.out.println("i is less than 5")
i is less than 5
```

Using the `if` statement, we can control the execution of `System.out.println("i is less than 5");`.

```
jshell> if (i < 5)
   ...> System.out.println("i is less than 5");
jshell>
```

The condition `i < 5` will evaluate to `false`, since `i` is currently `10`. Nothing is printed to console.

Let's change `i` to `4` and execute the `if` condition.

```
jshell> i = 4
i ==> 4
jshell> if (i < 5)
   ...> System.out.println("i is less than 5");
i is less than 5
jshell>
```

`i is less than 5` is printed to console.

By controlling the value stored in the variable `i`, we are able to control whether the statement `System.out.println("i is less than 5");` actually runs.

Hurrah! We just achieved conditional execution!

Just as we can compare a variable with a literal, it is possible to compare the values of two variables. The same set of comparison operators, namely `==`, `<`, `>`, `<=` and `>=` can be used.

```
jshell> int number1 = 5
number1 ==> 5
jshell> int number2 = 7
number2 ==> 7
jshell> if (number2 > number1)
   ...> System.out.println("number2 is greater than number1");
number2 is greater than number1

jshell> number2 = 3
number2 ==> 3
jshell> if (number2 > number1)
   ...> System.out.println("number2 is greater than number1");

jshell>
```

Summary

In this step, we:

- Understood the need for a conditionals

- Were introduced to the concept of logical expressions, and conditional operators
- Explored usage of the Java `if` statement

Step 24: Programming Exercise PE-04

1. Create four integer variables `a` , `b` , `c` and `d` . Write an `if` statement to print if the sum of `a` and `b` is greater than the sum of `c` and `d` .
2. Store three numerical values as proposed angles of a triangle in integer variables `angle1` , `angle2` and `angle3` . Create an `if` statement to state whether these three angles together can form a triangle.

   Hint: A triangle is a closed geometric figure with three angles, whose sum must exactly equal `180 degrees` .
3. Have a variable store an integer. Create an `if` statement to find out if it's an even number.

   Hint: Use operator `%` .

Step 25: Solution to PE-04

Solution 1

```
jshell> int a = 5
a ==> 5
jshell> int b = 7
b ==> 7
jshell> int c = 4
c ==> 4
jshell> int d = 3
d ==> 3
jshell> if (a + b > c + d)
   ...> System.out.println("a and b together tower above c plus d");
a and b together tower above c plus d

jshell>
```

Solution 2

```
jshell> int angleOne = 55
angleOne ==> 55
jshell> int angleTwo = 65
angleOne ==> 55
jshell> int angleThree = 60
angleOne ==> 55
jshell> if (angleOne + angleTwo + angleThree == 180)
   ...> System.out.println("The three angles together form a triangle");
The three angles together form a triangle

jshell> angleThree = 75
angleOne ==> 55
jshell> if (angleOne + angleTwo + angleThree == 180)
   ...> System.out.println("The three angles together form a triangle");

jshell>
```

Solution 3

```
jshell> int num = 10
num ==> 10
jshell> if (num % 2 == 0)
   ...> System.out.println("The number is even");
The number is even

jshell> num++
num ==> 11
jshell> if (num % 2 == 0)
   ...> System.out.println("The number is even");

jshell>
```

Step 26: `if` Statement again

Let's look at a few more examples of if statements.

Snippet-1

Here's a basic example of conditional execution.

```
jshell> int i = 5
i ==> 5
jshell> if (i == 5)
   ...> System.out.println("i is odd");
i is odd
```

Let's add two statements to the second line.

```
jshell> if (i == 5)
   ...> System.out.println("i is odd"); System.out.println("i is prime");
i is odd
i is prime

jshell>
```

Both text messages are printed out.

Let's change value of `i` to `6` and execute again.

```
jshell> i = 6
i ==> 6
jshell> if (i == 5)
   ...> System.out.println("i is odd"); System.out.println("i is prime");
i is prime

jshell>
```

`i is prime` is printed to console. Why?

Why is this statement executed when the condition is `false` ?

The `if` statement, by default, binds only to the next statement.

It does not control the execution of `System.out.println("i is prime");` , which in fact runs unconditionally, always.

Is there a way we can ensure conditional execution of a two statements? And more than two?

Snippet-2 : `if` with statement blocks

We seem to have pulled a rabbit out of the hat here, haven't we! The rabbit lies in the pair of *braces* : ' `{` ' and ' `}` '.

They are used to group a sequence of statements together, into a **block**. Depending on the condition value (`true` or `false`), either all the statements in this block are run, or none at all.

```
jshell> int i = 5
i ==> 5
jshell> if (i == 5) {
   ...> System.out.println("i is odd");
   ...> System.out.println("i is prime");
   ...> }
i is odd
i is prime

jshell> i = 6
i ==> 6
jshell> if (i == 5) {
   ...> System.out.println("i is odd");
   ...> System.out.println("i is prime");
   ...> }

jshell>
```

It is considered good programming practice to *always* use blocks in an `if` conditional. Here is an example:

```
if (i == 5) {
    System.out.println("i is odd");
}
```

A block can also consist of a single statement! This improves readability of code.

Summary

In this step, we:

- Saw the importance of using statement blocks with `if` conditionals
- Understood how control flow can be made more readable

Step 27: Introducing Loops: The `for` Statement

The *PMT-Challenge* needs us to print a total of 10 table entries. The `for` **loop** is a suitable iteration mechanism to get this done.

The word *loop* means exactly what the English dictionary says. *As `i` varies from `1` through `10` , do some stuff involving `i`*

`for` loop is built this way:

```
for(initialization; condition; update) {
    statement1;
    statement1;
}
```

Here's how above code runs:

1. Execute `initialization` code.
2. If `condition` evaluates to true, execute `statements` . Else, go out of loop.
3. Execute `update`
4. Repeat 2 and 3

We've already seen these components in isolation:

- *initialization* : `int i = 1`
- *condition* : `i <= 10`
- *update* : `i++`

Let's now put them together to get the bigger picture. Here is how it might look:

```
for (int i = 1; i <= 10; i++) {
    System.out.println("Hello World");
}
```

This loop, when executed, prints the message "Hello World" on a separate line, for a total of `10` times.

Snippet-1 : *PMT-Challenge* Solution

We need to replace the "Hello World" message with the console print of a table entry. This print is controlled by `i` taking values from `1` through `10` .

```
jshell> int i
i ==> 0
jshell> for (i=0; i<=10; i++) {
   ...> System.out.printf("%d * %d = %d", 5, i, 5*i).println();
   ...> }
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 2
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

Snippet-1 Explained

1. The first step is the initialization: `i = 1` .
2. After this, the statement `System.out.printf("%d * %d = %d", 5, i, 5*i).println();` is executed once.
3. Then, the update occurs: `i++` .
4. Immediately after, the condition `i<=10` is evaluated. It returns `true` . Progress happens. Since it is a `for` *loop*, the statement is executed again.
5. Thereafter, this sequence is executed until `i` gets a value of `11` (due to successive updates).
6. The moment `i` reaches `11` , , the condition becomes `false` . The looping within the `for` terminates.

Meanwhile, the Multiplication Table for `5` , for entries `1` through `10` has been displayed!

Now, wasn't that really elegant? It sure was!

Let's pat ourselves on the back for having reached this stage of learning. This elegant, yet powerful technique (*loops*) is used in almost every Java program that's written.

Summary

In this step, we:

- Observed why we need a looping construct, such as the `for`
- Understood the mechanism of a `for` loop
- Used a `for` loop in iteration, to solve our *PMT-Challenge*

Step 28: Programming Exercise PE-05

1. Repeat the entire process at arriving at the Multiplication Table Print problem, now for the number `7` . Start with a fresh `JShell` session, if you're still using an existing one for quite some time (Rinse, and repeat!).
2. Use the final solution to print Multiplication Tables for `6` and `10` .
3. Print the integers from `1` to `10` .
4. Print the integers from `10` to `1` .
5. Print the squares of the integers from `1` to `10` .
6. Print the squares of the first `10` even integers.
7. Print the squares of the first `10` odd integers.

Step 29: Solution to PE-05

Solution 2

```
    jshell> int i
    i ==> 0
    jshell> for (i=0; i<=10; i++) {
       ...> System.out.printf("%d * %d = %d", 6, i, 6*i).println();
       ...> }
    6 * 1 = 6
    6 * 2 = 12
    6 * 3 = 18
    6 * 4 = 24
    6 * 5 = 30
    6 * 6 = 36
    6 * 7 = 42
    6 * 8 = 48
    6 * 9 = 54
    6 * 10 = 60

    jshell> i = 0
    i ==> 0
    jshell> for (i=0; i<=10; i++) {
       ...> System.out.printf("%d * %d = %d", 10, i, 10*i).println();
       ...> }
    10 * 1 = 10
    10 * 2 = 20
    10 * 3 = 30
    10 * 4 = 40
    10 * 5 = 50
    10 * 6 = 60
    10 * 7 = 70
    10 * 8 = 80
    10 * 9 = 90
    10 * 10 = 100
```

Solution 3

```
    jshell> for (int i=1; i<=10; i++) {
```

```
   ...> System.out.printf(i).println();
   ...> }
1
2
3
4
5
6
7
8
9
10
```

Solution 4

This is the first time we are using `i--` . Isn't this interesting?

```
jshell> for (int i=10; i>0; i--) {
   ...> System.out.printf(i).println();
   ...> }
10
9
8
7
6
5
4
3
2
1
```

Solution 5

```
jshell> for (int i=1; i<=10; i++) {
   ...> System.out.printf(i*i).println();
   ...> }
1
4
9
16
25
36
49
64
81
100
```

Solution 6

`update` of a for loop can do a lot of things. Here, we are using `i += 2` .

```
jshell> for (int i=2; i<20; i += 2)
   ...> System.out.printf(i*i).println();
   ...> }
4
16
36
64
100
144
```

```
196
256
324
400
```

Solution 7

```
jshell> for (int i=1; i<=19; i += 2) {
   ...> System.out.printf(i*i).println();
   ...> }
1
9
25
49
81
121
169
225
289
361
```

Step 30: Puzzling You With `for`

In the conceptual form of the `for` construct:

```
for (initialization; condition; updation) {
    statement;
    statement;
    //...
    statement;
}
```

It may surprise you that each one of *initialization*, *condition*, *updation* and *statements block* is **optional**. They can all be left out individually, in combination, or all altogether!! Let's examine a few interesting cases in code.

1. Empty initialization, Empty Statement

Increments the control variable in quick succession until the condition evaluates to false.

```
jshell> int i = 1
i ==> 1
jshell> for (; i<=10; i++);
jshell> i
i ==> 11
jshell>
```

2. Multiple initializations, Empty Statement

You can have multiple statements in `initialization` and `update` separated by `,` .

```
jshell> int j
i ==> 11
jshell> for (i=1, j=2; i<=10; i++, j++);
jshell> i
i ==> 11
jshell> j
```

```
    j ==> 12
    jshell>
```

In the example below, i is incremented and j is decremented.

```
    jshell> for (i=1, j=2; i<=10; i++, j--);
    jshell> i
    i ==> 11
    jshell> j
    j ==> -8
    jshell>
```

3. Infinite Loop

An infinite loop is one where the *condition* is left *empty*. An empty condition always evaluates to `true`. Since a `for` loop only terminates when the condition becomes `false`, such a loop this never terminates.

It can only be terminated externally with a keyboard interrupt (`CTRL + c`).

```
    jshell> for (;;);

    ^C
    jshell>
```

4. Statement Block in for

As in case of the `if` conditional statement, we can have statement blocks in `for` loops as well. As before, we enclose the statement set between braces (' `{` ' and ' `}` '). The statements are executed repeatedly, in the same order as they appear in the block.

```
    jshell> for (i=1; i<=5; i++) {
       ...> System.out.println("No 1");
       ...> System.out.println("No 2");
       ...> }
    No 1
    No 2
    No 1
    No 2
    No 1
    No 2
    No 1
    No 2
    No 1
    No 2
```

Summary

In this step, we saw that all components of a `for` loop are optional:

- *initialization*
- *condition*
- *updation*
- *statement block*

Step 31: A Review Of Basic Concepts

Before we go further, let's quickly get a Big Picture of what we are doing here!

A **computer** is a *machine* that does a job for you and me. It is can be used to run tasks that we find complicated, time-consuming, or even boring! For instance, a laptop can play music from a CD, videos from the web, or fire a print job.

We have mobile phones around us, which are mini versions of computers. Then there are others, ranging from blood-sugar monitors to weather-forecast systems. Computers surround us, wherever we go!

Any computer is made up of two basic layers:

- The **hardware**: Consists of all the *electronic* and *mechanical* parts of the computer, including the *electronic circuits*.
- The **software**: Describes the *instructions* fed into the computer, which are stored and run on its *hardware*.

If the human body were a computer,

- Its *hardware* would consist of the various organs (such as limbs, blood and heart)
- Its *software* would be the signals from the nervous system, which drive these organs.

Computer programming involves writing software instructions to run tasks on a computer. The user who gives these instructions is called the **programmer**. Often, computer programming involves solving challenging, and very interesting problems.

In the previous steps, we introduced you to the basic Java language concepts and constructs.

We solved a programming challenge, the *PMT-Challenge* using basic Java constructs.

We used JShell REPL to learn the following concepts, Step by-step:

- Expressions, Operators and Statements
- Variables and Formatted Output
- Conditionals and Loops

At each step, we applied fresh knowledge to enhance our solution to the *PMT-Challenge*

Hope you liked the journey thus far. Next sections delve deeper into Java features, using the same Step by-step approach. We will catch up again soon, hopefully!

Understanding Methods

Feeling good about where you are right now? You just created an elegant, yet powerful solution to the *PMT-Challenge*, using:

- Operators, variables and expressions
- Built-in formatted output
- Conditionals for control-flow, and
- Iteration through loops

And guess what we ended up with? A good-looking display of a Multiplication Table! There's a good chance people in your circles want to see it, use it and maybe share it among their friends.

However, some might be unhappy that it works only for `5`, and not for `8` and `7`. Maybe it would, but then they would need to type in those lines of code again. This might disappoint them, and your new found popularity would slide downhill.

Surely a more elegant solution exists, a pattern waiting to unravel itself.

Exist it does, and the mechanism to use it lies with Java **methods**. A `method` is a feature that allows you to group together a set of statements, and give it a name. This name represents the *functionality* of that set, which can be re-used when necessary.

A method is essentially a *routine* that performs a certain task, and can do it any number of times it is asked to. It may also return a result for the task it performs. The syntax for a ***method definition*** is along these lines:

```
ReturnType   methodName () {
  method-body
}
```

Where

- `methodName` is the name of the routine
- `method-body` is the set of statements
- a pair of braces `{` and `}` enclose `method-body`
- `ReturnType` is the type of `methodName` 's return value

Summary

In this step, we:

- Examined the need for labeling code, and its reuse
- Learned that a Java method fits the bill
- Saw how a method definition looks like, conceptually

Step 01 : Defining A Simple Method

We will start off, by writing a simple method that prints " `Hello World` " twice on your screen.

```
    jshell> void sayHelloWorldTwice() {
       ...> System.out.println("Hello World");
       ...> System.out.println("Hello World");
       ...> }
    | created method sayHelloWorldTwice()
```

A little bit of theory:

- Above code for the method is called **method definition**.
- `void` indicates that the method does not return any computed result - We will examine return values from methods, a little later.

When the code ran, it didn't exactly welcome us twice, did it? All we got was a boring message from `JShell` , mumbling `created method sayHelloWorldTwice()` .

That's because defining a method ***does NOT*** execute its statement body.

Since the statement block is not stand-alone anymore, its functionality needs to be **invoked**. This is done by writing a **method call**.

Let's look at a few examples for **method calls**.

```
    jshell> sayHelloWorldTwice
    | Error:
    | cannot find symbol
    | symbol: variable sayHelloWorldTwice
```

```
| sayHelloWorldTwice
| ^-----------------^

jshell> sayHelloWorldTwice()
Hello World
Hello World
```

All that we had to do was to add parantheses to name of the method.

> The trailing ' `;` ' can be left out in `JShell`, and is another example of syntax rules being relaxed.

> Method names need to follow the same rules as variable names.

Summary

In this step we:

- Got our hands dirty coding our first method
- Learned that defining and invoking methods are two different steps

Step 02: Exercise-Set PE-01

Let's now reinforce our basic understanding of methods, by trying out a few exercises.

Exercise Set -5

1. Write and execute a method named `sayHelloWorldThrice` to print `Hello World` thrice.
2. Write and execute a method that prints the following four statements:

   `I've created my first variable`

   `I've created my first loop`

   `I've created my first method`

   `I'm excited to learn Java`

Solutions to PE-01

Solution-1

```
jshell> void sayHelloWorldThrice() {
   ...> System.out.println("Hello World");
   ...> System.out.println("Hello World");
   ...> System.out.println("Hello World");
   ...> }
| created method sayHelloWorldThrice()

jshell> sayHelloWorldThrice()
Hello World
Hello World
Hello World
```

Solution-2

```
jshell> void sayFourThings() {
   ...> System.out.println("I've created my first variable");
   ...> System.out.println("I've created my first loop");
   ...> System.out.println("I've created my first method");
   ...> System.out.println("I'm excited to learn Java");
   ...> }
| created method sayFourThings()

jshell> sayFourThings()
I've created my first variable
I've created my first loop
I've created my first method
I'm excited to learn Java
```

Step 03: Editing A Method Definition (`Jshell` Tips)

Snippet-01: Editing sayHelloWorldTwice()

```
jshell> void sayHelloWorldTwice() {
   ...> System.out.println("Hello World");
   ...> System.out.println("Hello World");
   ...> }
| created method sayHelloWorldTwice()
```

The `/methods` command lists out the methods defined in the current session.

```
jshell> /methods
| void sayHelloWorldTwice()
jshell>
```

The `/list` command lists the code of the specified method.

```
jshell> /list sayHelloWorldTwice
59 : void sayHelloWorldTwice() {
System.out.println("HELLO WORLD");
System.out.println("HELLO WORLD");
}
jshell>
```

The `/edit` command allows you to modify the method definition, in a separate editor window.

```
jshell> /edit sayHelloWorldTwice
| modified method sayHelloWorldTwice
jshell> sayHelloWorldTwice()
HELLO WORLD
HELLO WORLD
jshell>
```

The `/save` method takes a file name as a parameter. When run, it saves the session method definitions to a file.

```
    jshell> /save backup.txt
    jshell> /exit
    | Goodbye

in28minutes$>
```

Summary

In this step, we explored a few `JShell` tips that make life easier for you while defining methods

Step 04: Methods with Arguments

We wrote the method `sayHelloWorldTwice` to say `Hello World` twice. In the programming exercise, we wanted to print `Hello World` thrice and we wrote a new method `sayHelloWorldThrice` .

Imagine you're in the Java classroom, where your teacher wants to test your Java skills by saying: *"I want you to print **`Hello World`** an arbitrary number of times"*.

Now, that probably would test your patience as well!

How to write a method to print `Hello World` an arbitrary number of times?

The thing to note is the word "arbitrary", which means the method body should have no clue! This number has to come from outside the method, which can only happen with a method call.

External input to a method is given as a **method argument**, or *parameter*.

To support arguments during a call, the concept of a method needs to be tweaked to:

```
ReturnType methodName(ArgType argName) {

    method-body

}
```

The only addition to the concept of a method, is the phrase

`ArgType argName`

within the parentheses. `argName` represents the argument, and `ArgType` is its type. The next example should clear the air for you.

Snippet-01: Method with an argument: Definition

Let's look at a method definition using an argument `numOfTimes` .

```
jshell> void sayHelloWorld(int numOfTimes) {
   ...> }
| created method sayHelloWorld(int)
```

Let's try to call the method.

```
jshell> sayHelloWorld()
| Error:
| method sayHelloWorld in class cannot be applied to given types;
| required : int
| found : no arguments
| reason : actual and formal argument lists differ in length
| sayHelloWorld(
|^-----------------^
jshell> sayHelloWorld(1)
jshell>
```

Method call must include the same number and types of arguments, that it is defined to have. `sayHelloWorld()` is an error because `sayHelloWorld` is defined to accept one parameter. `sayHelloWorld(1)` works.

However, the method does not do anything. Isn't it sad?

Snippet-02 : Passing and accessing a parameter

The next example will show you how method body code can access arguments passed during a call. Not only that, the argument values can be used during computations.

```
void sayHelloWorld(int numOfTimes) {
    System.out.println(numOfTimes);
}
```

`System.out.println(numOfTimes)` prints the value of argument passed.

A method can be invoked many times within a program, and each time different values could be passed to it. The following example will illustrate this fact.

```
jshell> sayHelloWorld(1)
1
jshell> sayHelloWorld(2)
2
jshell> sayHelloWorld(4)
4
jshell> /edit sayHelloWorld
| modified method sayHelloWorld(int)
jshell>
```

Snippet-03: More complex method

Code inside a method can be any valid Java code! For instance, you could write code for iteration, such as a `for` loop.

Let's look at an example:

```
void sayHelloWorld(int numOfTimes) {
    for (int i=1; i<=numOfTimes; i++) {
        System.out.println(numOfTimes);
    }
}
```

In the above example, we printed `numOfTimes` a total of `numOfTimes` for each method call.

We print `2` two times and `4` four times.

```
jshell> sayHelloWorld(2)
2
2
jshell> sayHelloWorld(4)
4
4
4
4
jshell>
```

Snippet-4 : Saying "Hello World", Again and again...

We wanted to print "Hello World" multiple times. Let's update the method:

```
void sayHelloWorld(int numOfTimes) {
    for (int i=1; i<=numOfTimes; i++) {
        System.out.println("Hello World");
        }
    }
```

Isn't this cool?

```
    jshell> sayHelloWorld(1)
    Hello World
    jshell> sayHelloWorld(2)
    Hello World
    Hello World
    jshell> sayHelloWorld(4)
    Hello World
    Hello World
    Hello World
    Hello World
    jshell>
```

You can now proudly demonstrate this code to your Java instructor. Your program can print "Hello World" an arbitrary number of times!

What started off giving you a headache, will probably keep you in her good books, for the rest of your course!

Armed with this confidence booster, let's now see how Java treats mistakes you may make. .

Snippet-5 : Parameter type mismatch

Java is a strongly typed language, with strict rules laid out for type compatibility. We saw that with variables, and how they play out with expressions and assignments. The same type compatibility rules are enforced by the compiler, when it needs to match the arguments from method calls with method definition.

```
    jshell> sayHelloWorld("value")
    | Error:
    | incompatible types: java.lang.String cannot be converted to int
    | sayHelloWorld("value")
    |               ^-----^
    jshell> sayHelloWorld(4.5)
    | Error:
    | incompatible types: possibly lossy conversion from double to int
    | sayHelloWorld(4.5)
    |               ^-^
    jshell>
```

Summary

In this step, we:

- Understood why Java supports method arguments, and how we may use them
- Observed how method arguments lead to convenience and reuse
- Decided to abide by type compatibility for actual arguments

Step 05: Exercise Set PE-02 (With Solutions)

Exercises

1. Write a method `printNumbers(int n)` that prints all successive integers from `1` to `n`.
2. Write a method `printSquaresOfNumbers(int n)` that prints the squares of all successive integers from `1` to `n`.

Solution-01

```
jshell> void printNumbers(int n) {
   ...> for(int i=0; i< n; i++) {
   ...> System.out.println(i);
   ...> }
   ...> }
| created method printNumbers(int)
```

jshell>

Solution 2

```
jshell> void printSquaresOfNumbers(int n) {
   ...> for(int i=0; i< n; i++) {
   ...> System.out.println(i*i);
   ...> }
   ...> }
| created method printSquaresOfNumbers(int)
jshell>
```

Step 07: *PMT-Challenge* Revisited (And Some Puzzles)

A method, in its own right, provides an elegant way to name and reuse a code block. In addition, its behavior can be controlled with parameters.

Can we top-up the existing solution to the *PMT-Challenge*, with even more elegance? You're right, we're talking about:

1. Writing a method to print the multiplication table for 5.
2. Using this method to print the multiplication table for any number.

Want to take a look? Dive right in.

Here's what we did earlier:

```
jshell> for (int i=1; i<=10; i++) {
   ...> System.out.printf("%d * %d = %d", 5, i, 5*i).println();
   ...> }
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

Let's wrap this in a method:

```
void printMultiplicationTable() {
    for (int i=1; i<=10; i++) {
    System.out.printf("%d * %d = %d", 5, i, 5*i).println();
    }
}
```

You can call it to print 5 table.

```
jshell> printMultiplicationTable()
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
jshell>
```

Summary

In this step, we:

- Revisited the *PMT-Challenge* solution we had earlier
- Enclosed its logic within a method definition, `printMultiplicationTable()`
- Haven't fully explored its power yet!

Step 08: Methods With Arguments, And Overloading

The real power of a definition such as `printMultiplicationTable()` is when we arm it with arguments. Not verbal arguments, but "value" ones. That's right!

We will modify `printMultiplicationTable()` to have it accept a parameter, which would make it more flexible.

```
void printMultiplicationTable(int number) {
    for (int i=1; i<=10; i++) {
        System.out.printf("%d * %d = %d", number, i, number*i).println();
    }
}
```

We can now print the multiplication table of any number, by calling the method `printMultiplicationTable(number)`.

```
jshell> printMultiplicationTable(6)
6 * 1 = 6
6 * 2 = 12
6 * 3 = 18
6 * 4 = 24
6 * 5 = 30
6 * 6 = 36
6 * 7 = 42
6 * 8 = 48
6 * 9 = 54
6 * 10 = 60
jshell>
```

Nothing new to explain here. We have only combined stuff you have been learning so far, most recently adding the flavor of methods to the existing code.

Soak in the power, simplicity and elegance of this program, folks!

Overloaded Methods

It turns out we can call both versions of the method, `printMultiplicationTable()` and `printMultiplicationTable(5)` :

```
jshell> printMultiplicationTable()
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
jshell>
```

and

```
jshell> printMultiplicationTable(5)
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
jshell>
```

You can have multiple methods with same name but different number of parameters. This is called ***method overloading***.

Summary

In this step, we:

- Enhanced the logic of `printMultiplicationTable()` by passing arguments
- Added flexibility to the *PMT-Challenge* solution
- Were introduced to the concept of method overloading

Step 09: Methods With Multiple Arguments

It is possible, and pretty useful, to define methods that accept more than one argument.

We have already seen how to call two in-built Java methods, `Math.max` and `Math.min` , which accept 2 arguments each.

Time now to enrich our understanding, by writing one such method ourselves as well.

A method `void sum(int, int)` that computes the sum of two integers, and prints their output.

```
jshell> void sum(int firstNum, int secondNum) {
   ...> int sum = firstNum + secondNum;
   ...> System.out.println(sum);
   ...> }
| created method sum(int, int)
```

```
jshell> sum(5, 10)
15
jshell>
```

A method `void sum(int, int, int)` that computes the sum of three integers, and prints their output.

```
jshell> void sum(int firstNum, int secondNum, int thirdNum) {
   ...> int sum = firstNum + secondNum + thirdNum;
   ...> System.out.println(sum);
   ...> }
| created method sum(int,int,int)
jshell> sum(5, 10, 15)
30
```

There are overloaded methods. They have same name `sum` and have different number of arguments.

Summary

In this step, we:

- Used our understanding of method overloading to define our own methods

Step 10: Returning From A Method

A method can also be coded to return the result of its computation. Such a result is called a **return value**.

A lot of built-in Java methods have return values, the most notable we have seen being `Math.min()` and `Math.max()` .

```
jshell> Math.max(15, 25)
$1 ==> 25
jshell> $1
$1 ==> 25
```

`Math.max()` does return a value, which is stored in the `JShell` variable `$1` .

A return mechanism is quite important, as it provides you the flexibility of:

- Sharing computed results with other code and methods
- Improving the breaking down of a problem, into sub-problems

The next example explains how you can collect a method's return value.

```
jshell> int max = Math.max(15, 25)
max ==> 25
jshell> max
max ==> 25
jshell>
```

We are storing the return value in a variable `int max` .

We could define our own method that returns a value, in a similar manner. In the method `sumOfTwoNumbers` above, instead of displaying `sum` at once, we could `return` it to the calling-code.

```
jshell> int sumOfTwoNumbers(int firstNum, int secondNum) {
   ...> int sum = firstNum + secondNum;
   ...> return sum;
   ...> }
| created method sumOfTwoNumbers(int,int)
jshell> sumOfTwoNumbers(1, 10)
$2 => 11
```

The statement `return sum;` is called a **return statement**.

When the code `sum = sumOfTwoNumbers(1, 10)` is run, `sum` collects the result returned by `sumOfTwoNumbers()`. You could now print `sum` on the console, store it in another variable, or even invoke a different method by passing it as an argument!

```
jshell> sum = sumOfTwoNumbers(1, 10)
sum => 11_
jshell> sum = sumOfTwoNumbers(15, 15)
sum => 30
jshell>
```

Summary

In this step, we:

- Understood the need for a return mechanism with methods
- Explored the syntax for a return statement
- Wrote our own method `sumOfTwoNumbers()` with a return value, and saw how it became more flexible

Step 11: Exercise-Set PE-04 (With Solutions)

Let's now try a few exercises on methods that return values.

Exercises

1. Write a method that returns the sum of three integers.
2. Write a method that takes as input two integers representing two angles of a triangle, and computes the third angle. *Hint: The sum of the three angles of a triangle is 180 degrees*.

Solution-01

```
jshell> int sumOfThreeNumbers(int firstNum, int secondNum, int thirdNum) {
   ...> int sum = firstNum + secondNum + thirdNum;
   ...> return sum;
   ...> }
| created method sumOfThreeNumbers(int,int, int)
jshell>
```

Solution-02

```
jshell> int getThirdAngle(int firstAngle, int secondAngle) {
   ...> int sumOfTwo = firstAngle + secondAngle;
   ...> return 180 - sum;
   ...> }
| created method getThirdAngle(int,int)
jshell>
```

Step 12: Java Methods, A Review

In this section, we introduced you to the concept of methods in Java.

- We understood that methods are routines that perform a unit of computation. They group a set of statements together into a block, and need to be run by being invoked, through a method call.
- We can define methods that accept no arguments, a single argument and multiple arguments.
- A method can also be defined to return results of its computation, and this makes it your program more flexible.

Understanding the Java Platform

JShell is amazing to start with the basics of Java programming. It abstracts all complexities behind writing, compiling and running Java code.

What's happening behind the screens? Let's find out.

Step 01: The Java Platform - An Introduction

A computer only understands binary code, which are sequences of `0` ’s and `1` ’s (called **bits**). All instructions to a computer should be converted to `0` ’s and `1` ’s before execution.

When we wrote our code in JShell, how was it converted to binary code?

We write our programs in a high-level language, such as Java, as it is easier to learn, remember and maintain.

Who converts it to binary code?

Typically, **Compiler** is a program which understands the syntax of your programming language and converts it into binary code.

Java designers wanted it to be Platform independent. Compile the code once and run it anywhere.

However, different Operating Systems have different instruction sets - different binary code.

Java provides an interesting solution:

- All Java compilers translate source code to an **intermediate representation** (**bytecode**).
- To run Java programs (**bytecode**), you need a ***JVM*** (Java Virtual Machine).
  - ***JVM*** understands **bytecode** and runs it.
  - There are different JVM's for different operating systems. Windows JVM converts **bytecode** into Windows executable instructions. Linux JVM converts **bytecode** into Linux executable instructions.

The Java compiler translates Java source code to **bytecode**, which is stored as a **.class** file on the computer.

Summary

In this step, we:

- Understood the role of the Java compiler in translating source code
- Realized the need for an Intermediate Representation (IR) such as bytecode
- Understood how the (Compiler + JVM + OS) combination ensure source code portability

Step 02: Creating a Java `class`

First, let's understand the concept of a `class` .

Let's consider an example:

A **Country** is a *concept*. **India**, **USA** and **Netherlands** are *examples* or *instances* of **Country**.

Similarly, a `class` is a template. `Objects` are `instances` of a `class` .

We will discuss more about `class` and `object` in the section on Object Oriented Programming.

The syntax to create a class, and its objects, is very simple.

```
jshell> class Country {
   ...> }
created class Country
```

Let's create an instance of the class:

```
jshell> Country india = new Country();
india ==> Country@6e06451e
```

The syntax is simple - `ClassName objectName = new ClassName()` .

`india` is an object of type `Country` , and is stored in memory at a location indicated by `6e06451e` .

Let's create a few more instances of the class:

```
jshell> Country usa = new Country();
usa ==> Country@6e1567f1
jshell> Country netherlands = new Country();
netherlands ==> Country@5f134e67
jshell>
```

`india` , `usa` and `netherlands` are all different objects of type `Country` .

We actually left the contents of the `class` `Country` bare.

A `class` does often include both data (member variables) and method definitions. More on this at a relevant time.

Let's consider another example:

```
jshell> class Planet {
   ...>}
created class Planet
jshell> Planet planet = new Planet();
planet ==> Planet@56ef9176
jshell> Planet earth = new Planet();
earth ==> Planet@1ed4004b
jshell> Planet planet = new Planet();
venus ==> Planet@25bbe1b6
```

We can also create a `class` for a different concept, like a `Planet` .

`Planet` is now a different template. `planet` , `earth` and `venus` are instances of `Planet` class.

Summary

In this step, we:

- Saw how `class` creation actually creates a new type
- Understood how to create instances of classes - objects

Step 03: Adding A Method To A `class`

A method defined within a `class` , denotes an action than can be performed on objects of that `class` .

Let's define a method inside `Planet` - to `revolve()` !

```
 jshell> class Planet {
...> void revolve() {
...> System.out.println("Revolve");
...> }
...> }
replaced class Planet
update replaced variable planet, reset to null
update replaced variable earth, reset to null
update replaced variable venus, reset to null
```

Syntax of `revolve()` is similar to other methods we've created before.

> The `class` template got updated as a result of adding new method. Earlier instances `planet` , `earth` and `venus` got reset to `null` as they are based on the old template

Let's re-instantiate a few objects.

```
jshell> Planet earth = new Planet();
earth ==> Planet@192b07fd
jshell> Planet planet = new Planet();
venus ==> Planet@64bfbc86
jshell>
```

How do we call the `revolve` method?

```
jshell> Planet.revolve();
| Error:
| non-static method revolve() cannot be referenced from a static context
| Planet.revolve();
|^-----------------^
jshell> earth.revolve();
Revolve
jshell> venus.revolve();
Revolve
jshell>
```

Our attempt to perform `revolve()` did not work with the syntax `Planet.revolve()` as it is not a static method (more on that in a later section).

Invoking `revolve()` through syntax such as `earth.revolve()` succeeds, because `earth` is an object of type `Planet` .

Summary

In this step, we:

- Learned how to add a method definition to an existing `class`
- Discovered how to invoke it, on objects of the `class`

Step 04: Writing and Running Java Code in separate Source Files

So far, we have enjoyed creating classes and defining methods within them, all with our friendly neighborhood `JShell` .

`JShell` kind of spoiled us kids, by relaxing some Java syntax rules, invoking the compiler on our behalf, and also running the code like room service.

The time has come for us frog princes to outgrow the proverbial well. Yes, you're right! Time to start writing code in a proper code editor, and smell the greener grass on that side.

Let's start with creating a new source code file - 'Planet.java'. Choose any folder on your hard-disk and create the file shown below:

Planet.java

```
class Planet {
    void revolve() {
        System.out.println("Revolve");
    }
}
```

> You can get the code that we wrote earlier in JShell for `Planet` by running the command `/list Planet`, and copying the code.

> You can use any text editor of your choice, for now.

The source file name must match the `class` name it contains. Here, we must name the file `Planet.java`.

The next course of action would be to compile this code (Stage 2). For that, exit out of `JShell`, and run back to your system terminal or command prompt. Now, see what plays out!

```
jshell> /exit
|  Goodbye
```

`cd` to the folder where you have created the file `Planet.java`

```
in28minutes$> cd in28Minutes/git/JavaForBeginners/
command-prompt> ls
Planet.java
```

Check Java version.

```
command-prompt> java -version
java version "x.0.1"
Java(TM) SE Runtime Environment (build x.0.1+11)
Java HotSpot(TM) 64-bit Server VM (build x.0.1+11, mixed mode)
```

You can compile java code using `javac Planet.java`. You can see that a new file is created `Planet.class`. This contains your `bytecode`.

```
command-prompt> javac Planet.java
command-prompt> ls
Planet.class         Planet.java_
```

You can run the class using command `java Planet`.

```
command-prompt> java Planet
Error: Main method not found inside class Planet, please define the main method as
public static void main(String[] args)
or a JavaFX application class must extend javax.application.Application
command-prompt>
```

Why is there an error? Let's discuss that in the next step.

Summary

In this step, we:

- Ventured out of `JShell` into unexplored territory, to write source code
- Learned how to invoke the compiler to compile source files, from the terminal

Step 05: Introducing `main()` , And Running Bytecode

In the previous step, we compiled ***Planet.java***. In this step, let's run the bytecode generated, which lies inside ***Planet.class***.

Snippet-01: Running `Planet`

Console Commands

```
command-prompt> ls
Planet.class         Planet.java
command-prompt> java Planet
Error: Main method not found inside class Planet, please define the main method as
public static void main(String[] args)
or a JavaFX application class must extend javax.application.Application
command-prompt>
```

The code may have compiled without any errors, but what's the use if we cannot run the program to see what stuff it's got!

The `main()` method is essential to run code defined within any `class` .

The definition of `main()` needs to have this exact synatx:

`public static void main(String[] args) { /* <Some Code Goes In Here> */ }`

A few details:

- `void` is its return type
- `main` is its name
- `args` is its formal argument. Here, it's an **array** of `String` .
- `public and static``` are reserved Java keywords. More on these later sections.

Let's add one such definition within the `Planet` code. Open up your text editor and add the `main` method as shown below.

Planet.java

```
class Planet {
    void revolve() {
        System.out.println("Revolve");
    }
    public static void main(String[] args) {

    }
}
```

Let's run it now:

```
command-prompt> java Planet
Error: Main method not found inside class Planet, please define the main method as
public static void main(String[] args)
or a JavaFX application class must extend javax.application.Application
```

Why is there no change in result?

After changing your Java source file, you need to compile the code again to create a new class file.

```
command-prompt> javac Planet.java
command-prompt> ls
Planet.class         Planet.java_
```

Let's try it again.

```
command-prompt> java Planet
command-prompt>
```

We got a blank stare from the terminal. Why?

What did we write in the `main` method? Nothing.

```
    public static void main(String[] args) {

    }
```

Let's fix it. We need to edit ***Planet.java*** once again!

Planet.java

```
class Planet {
    void revolve() {
        System.out.println("Revolve");
    }
    public static void main(String[] args) {
        Planet earth = new Planet();
        earth.revolve();
    }
}
```

Console Commands

```
command-prompt> javac Planet.java
command-prompt> ls
Planet.class         Planet.java
command-prompt> java Planet
Revolve
command-prompt>
```

Bingo! We finally got what we wanted to see!

Summary

In this step, we:

- Learned that we need a `main()` method to run a Java program
- Discovered that after every code update, we need to compile that source file

Step 06: Puzzles About `Planet`

In this step, let's play with `Planet` source code. Get ready for a comedy of errors, by trial-and-error!

Below is the source code for `Planet` that worked so well for us:

Planet.java

```
class Planet {
    void revolve() {
        System.out.println("Revolve");
    }

    public static void main(String[] args) {
        Planet earth = new Planet();
        earth.revolve();
    }
}
```

Here starts our nut-job.

Snippet-01: Messing-up `main()` - v1

Change method name from `main` to `main1`

Planet.java

```
class Planet {
    void revolve() {
        System.out.println("Revolve");
    }

    public static void main1(String[] args) {
        Planet earth = new Planet();
        earth.revolve();
    }
}
```

Console Commands

```
command-prompt> javac Planet.java
command-prompt> java Planet
Error: Main method not found inside class Planet, please define the main method as
public static void main(String[] args)
or a JavaFX application class must extend javax.application.Application
command-prompt>
```

Don't mess with the method-name of `main()` .

Other parts also need to remain the same, such as:

- The return-type: `void`
- Argument type: `String[]`
- The presence of keywords `public` and `static` .

Snippet-02: Messing-up `main()` - v2

Remove semicolons after statements `Planet earth = new Planet()` and `earth.revolve()` .

Planet.java

```
class Planet {
    void revolve() {
        System.out.println("Revolve");
    }

    public static void main(String[] args) {
        Planet earth = new Planet()
        earth.revolve()
    }
}
```

Console Commands

```
command-prompt> javac Planet.java
Planet.java:6: error: ';' expected
Planet earth =  new Planet()
|_______________________^
Planet.java:7: error: ';' expected
earth.revolve()
|_____________^
2 errors
command-prompt>
```

The `;` symbol is the Java statement separator, which is mandatory.

It is not needed in JShell but mandatory when you write separate Java source files.

Snippet-03: Messing-up `main()` - v3

Make a typo `Planet earth = new Plane();`

Planet.java

```
class Planet {

    void revolve() {
        System.out.println("Revolve");
    }

    public static void main(String[] args) {
        Planet earth = new Plane();
        earth.revolve();
    }
}
```

Console Commands

```
command-prompt> javac Planet.java
Planet.java:6: error: cannot find symbol
Planet earth =  new Plane();
|________________^
symbol: class Plane
location: class Planet
```

```
1 error
command-prompt>
```

We misspelled `Planet` as **`Plane`** , and the compiler trapped this error. It shows us a red flag with a message to help us correct our mistake.

Snippet-04: Messing-up `main()` - v4

Call a non existing method `earth.revolve1();` .

Planet.java

```
class Planet {
    void revolve() {
        System.out.println("Revolve");
    }

    public static void main(String[] args) {
        Planet earth = new Planet();
        earth.revolve1();
    }
}
```

Console Commands

```
command-prompt> javac Planet.java
Planet.java:7: error: cannot find symbol
earth.revolve1();
|_____^
symbol: method revolve1()
location: variable earth of type Planet
1 error
command-prompt>
```

We misspelled the name of the `revolve()` method, to **`revolve1()`** . Rightly, got rapped on our knuckles by the compiler.

Summary

In this step, we:

- Learned that the best way to learn about language features is to play around with working code
- Observed that `JShell` simplifies coding for us, even relaxing some syntax rules
- Concluded that developing Java code in code-editors is an empowering feeling!

Step 07: The JVM, JRE And JDK

What are *JVM*, *JRE* and the *JDK*?

Apart from the fact that all start with a '*J*', there are important differences in what they contain, and what they do.

The following list gives you a bird's eye view of things.

The **JVM** runs your program bytecode.

JRE = JVM + Libraries + Other Components

- *Libraries* are built-in Java utilities that can be used within any program you create. `System.out.println()` was a method in `java.lang` , one such utility.

- *Other Components* include tools for debugging and code profiling (for memory management and performance).

JDK = **JRE** + **Compilers** + **Debuggers**

- *JDK* refers to the **Java Development Kit**. It's an acronym for the bundle needed to compile (with the compiler) and run (with the *JRE* bundle) your Java program.

An interesting way to remember this organization is:

- **JDK** is needed to **Compile and Run** Java programs
- **JRE** is needed to **Run** Java Programs
- **JVM** is needed to **Run Bytecode** generated from Java programs

Let's check a couple of scenarios to test your understanding.

Scenario 1: You give *Planet.class* file to a friend using a different operating system. What does he need to do to run it?

First, install **JRE** for his operating system. Run using *java Planet* on the terminal. He does not need to run *javac*, as he already has the bytecode.

Scenario 2: You gave your friend the source file *Planet.java*. What does he need to do to run it?

Install *JDK* of a compatible version. Compile code using *javac Planet.java*. Run *java Planet* on his terminal.

In summary

- **Application Developers** *need* **==> JDK**
- **Application Users** *need* **==> JRE**

Summary

In this step, we:

- Understood the differences in the scope-of-work done by the JVM, JRE and JDK
- Realized how each of them could be invoked by different terminal commands, such as *javac* and *java*.

Eclipse

TODO - Need Revision

Introducing Java Software Development with Eclipse

- Features of the Eclipse IDE (Integrated Development Environment)
    - Installation & Configuration
    - Workspace Creation & Configuration
    - Project Creation & Organization
        - JRE Version Selection and Included Libraries
        - Eclipse Java Perspectives
        - Source Files Organization into Folders
        - Packages?
    - IDE User Interface Description
        - Perspective
        - Views

    - Console Content and Display Options

- Creating a new Java `class` in Eclipse (And related source file)

  - Package name

    - A Java solution to solving a problem could be composed of several application components. It is considered good programming arctice to identify a class for each distinct component. Package is the Java way to organize classes in source code.
    - Analogy: Eatables in a Refrigerator (Freezer, Chill-Tray, Vegetable-Tray, Bottle-Rack)

  - Public method stub : can be used to create a default `public static void main(String[] args)` signature

  - (Include Snapshots)

    - Class creation pop-up, with selected options
    - Default generated source code in editor window

  - Customizing the generated source code to our needs

    - Syntax Highlighting
      - Keywords
      - Built-in Types
      - Constant literals: numbers, strings
    - Auto-suggest feature of eclipse as code is being typed in

  - The "Run as --> Java Application" Option to compile-and-run the source code

    - Option is avaibale only for classes that have a main() method

Using Eclipse in Debug Mode

Execution of the Multiplication Table Program can be done Step by-step. That is, the entire application dies not execute at one go, printing the final output onto the console. We can stall its flow at several steps, by taking control of its execution using the Eclipse IDE. This is possible in a special mode of Eclipse, called **Debug Mode**.

The process is called **Debugging**, because this mode is heavily used not just to have fun seeing what happens, but also to detect and fix software **bugs**. A bug is a defect in the program being written, that leads to its incorrect execution. The process of removing bugs is called debugging. Humans being humans, programming errors are but natural, and a debug mode in the IDE is worth its wight in gold to the programmer. Here is how Eclipse facilitates debugging of Java applications.

- Prior to debugging application, we need to set few **Break-Points**. A Break-Point is a statement in the source of the program, before which execution under debug mode is suspended, and control is passed to the programmer.

- The overall state of the data in the suspended program is available to the programmer at that particular break-point. He/she gets to specify one or more break-points in the program, and when the execution is suspended, a list of possible actions can be performed, including:

  - Reading & modifying data variables

  - Resuming program execution

  - Re-executing current statement

  - Skipping all statements till the next break-point

    and others.

- The Eclipse IDE provides a very user-friendly and intuitive interface to the programmer for controlling execution of a program in Debug Mode (Provide Snapshots of IDE at various stages of Debug Mode execution).
  - Setting and Removing a Break-point (also Toggling)
  - Stepping over a method call, or into and out of a method body (during a method call)
    - Stepping into `int print()` and `int print(int)` and `int print(int,int,int)` gives us interesting infromation, but stepping into `System.out.printf` can freak you out! So, you may want to step over it.
  - For nested method calls, examine the method call **Stack Trace** (Call child, call parent relationship)
    - Example : `int print()` , `int print(int)` and `int print(int,int,int)` method call chain of `class MultiplicationTable` .
  - Viewing and Modifying current state of data variables, at a break point
  - Stepping through from one statement to another in the source code
    - As we step through, observe the view displaying changes in data variable values
    - Example of `for` loop control variable `i` , inside method `int print(int,int,int)` of `class MultiplicationTable`
    - Observe that execution exits from the `for` loop when the condition `i<=10` is no longer `true` .
  - Re-executing a line of code
  - Skipping execution of all statements till next break-point
  - Ignoring all remaining breakpoints, resume execution of code till completion

Eclipse IDE Keyboard Shortcuts

- Code Text Editor Shortcuts
- New Project/Class Creation Shortcuts
- Search for a Class

Differences between JShell and IDE

Variables just declared, but used without an initialization, will flag a compilation error in an IDE! But not so, it seems, in JShell. In regular software, variable initialization at point of declaration (called a "defintion", as we already know) is mandatory.

Snippet-1 : JShell redefines variables on demand

```
jshell> int i = 2
$1 ==> 2
jshell> int i = 4
$2 ==> 4
jshell> long i = 1000
$3 ==> 1000
jshell>
```

When the following code (similar to the one given to JShell) is compiled manually or by IDEs such as Eclipse, it will flag errors!

```
package com.in28minutes.firstjavaproject;

public class RedefineTestRunner {
    public static void main(String[] args) {
        int i = 2;
        int i = 4;
        long i = 1000;
        System.out.println(i);
    }
}
```

That is because Java source code is governed by strict **Scope Rules**.

Let's have another look at where we have reached with our solution, to the *PMT-Challenge* problem. Only now, let's change the code arrangement.

Snippet-01: Revisited - The *PMT-Challenge*

MultiplicationTable.java

```
    package com.in28minutes.firstjavaproject;
    public class MultiplicationTable {
        public static void print() {
            for(int i=1; i<=10;i++) {
                System.out.printf("%d * %d = %d", 5, i, 5*i).println();
            }
        }
    }
```

MultiplicationRunner.java

```
    package com.in28minutes.firstjavaproject;

    public class MultiplicationRunner {
        public static void main(String[] args) {
            MultiplicationTable table = new MultiplicationTable();
            table.print();
    }
}
```

Console Output

*5 * 1 = 5*

*5 * 2 = 10*

*5 * 3 = 15*

*5 * 4 = 20*

*5 * 5 = 25*

*5 * 6 = 30*

*5 * 7 = 35*

*5 * 8 = 40*

*5 * 9 = 45*

*5 * 10 = 50*

Snippet-01 Explained

- We have now split the code into two source files:
  - ***MultiplicationTable.java***: Houses the `MultiplicationTable` `class` definition, with some methods we need.
  - ***MultiplicationRunner.java***: Acts as the client of the `MultiplicationTable` `class`, to invoke its functionality. It defines a `main()` method, that instantiates a `MultiplicationTable` object, and invokes its

`print()` method.

- After this code was rearranged, we still got it to work!

Print-Multiplication-Table: Enhancements

- We now want to enhance our current solution to this challenge, even though we've come a long way. **"Once you get the whiff of success, sky is the limit!"**. Here is the changes we need:
  - Pass the number whose table needs to be printed, as an argument
  - Pass the (continuous) range of numbers, whose index entries in the table are to be printed. (For example, printing the table entries for `6` , with entries for indexes between `15` to `30`).

One way to achieve all this, would involve overloading the `print()` method. The next example is one such attempt.

Snippet-02: print() overloaded

Overloading `print()` works for us, and we now support three ways in which to display any table:

```
    public static void print() {
        for(int i=1; i<=10;i++) {
            System.out.printf("%d * %d = %d", 5, i, 5*i).println();
        }
    }
```

Console Output

*5 * 1 = 5*

*5 * 2 = 10*

*5 * 3 = 15*

*5 * 4 = 20*

*5 * 5 = 25*

*5 * 6 = 30*

*5 * 7 = 35*

*5 * 8 = 40*

*5 * 9 = 45*

*5 * 10 = 50*

The default table for `5` , with entries fixed within `1` to `10` .

```
    public static void print(int number) {
        for(int i=1; i<=10;i++) {
            System.out.printf("%d * %d = %d", number, i, number*i).println();
        }
    }
```

Console Output

*8 * 1 = 8*

*8 * 2 = 16*

*8 * 3 = 24*

*8 * 4 = 32*

*8 * 5 = 40*

*8 * 6 = 48*

*8 * 7 = 56*

*8 * 8 = 64*

*8 * 9 = 72*

*8 * 10 = 80*

Printing a table for any number, but with entries fixed between `1` and `1`.

```
	public static void print(int number, int from, int to) {
		for(int i=from; i<=to;i++) {
			System.out.printf("%d * %d = %d", number, i, number*i).println();
		}
	}
```

Console Output

*6 * 11 = 66*

*6 * 12 = 72*

*6 * 13 = 78*

*6 * 14 = 84*

*6 * 15 = 90*

*6 * 16 = 96*

*6 * 17 = 102*

*6 * 18 = 108*

*6 * 19 = 114*

*6 * 20 = 120*

Displaying a table for any number, with entries for any range

The full code:

MultiplicationTable.java

```
package com.in28minutes.firstjavaproject;

public class MultiplicationTable {
	public static void print() {
		for(int i=1; i<=10;i++) {
			System.out.printf("%d * %d = %d", 5, i, 5*i).println();
		}
	}

	public static void print(int number) {
        for(int i=1; i<=10;i++) {
            System.out.printf("%d * %d = %d", number, i, number*i).println();
        }
    }

    public static void print(int number, int from, int to) {
        for(int i=from; i<=to;i++) {
            System.out.printf("%d * %d = %d", number, i, number*i).println();
        }
    }
}
```

MultiplicationRunner.java

```
package com.in28minutes.firstjavaproject;

public class MultiplicationRunner {
    public static void main(String[] args) {
        MultiplicationTable table = new MultiplicationTable();
        table.print();
        table.print(8);
        table.print(6, 11, 20);
    }
}
```

Issue: Code Duplication In Print-Multiplication-Table

There is an issue with the code for `class MultiplicationTable`. In the final print format, what if we are asked to replace the multiplication symbol ' * ' with an ' X ', so that school kids like it better? Since there are three calls to `System.out.printf()`, one in each `print()` version, we make three changes.

Snippet-3: Code Duplication

MultiplicationTable.java

```
package com.in28minutes.firstjavaproject;

public class MultiplicationTable {
    public static void print() {
        for(int i=1; i<=10;i++) {
            System.out.printf("%d X %d = %d", 5, i, 5*i).println();
        }
    }

    public static void print(int number) {
        for(int i=1; i<=10;i++) {
            System.out.printf("%d * %d = %d", number, i, number*i).println();
        }
    }

    public static void print(int number, int from, int to) {
        for(int i=from; i<=to;i++) {
            System.out.printf("%d X %d = %d", number, i, number*i).println();
        }
    }
}
```

Console Output

5 X 1 = 5

```
5 X 2 = 10

5 X 3 = 15

5 X 4 = 20

5 X 5 = 25

5 X 6 = 30

5 X 7 = 35

5 X 8 = 40

5 X 9 = 45

5 X 10 = 50

8 * 1 = 8

8 * 2 = 16

8 * 3 = 24

8 * 4 = 32

8 * 5 = 40

8 * 6 = 48

8 * 7 = 56

8 * 8 = 64

8 * 9 = 72

8 * 10 = 80

6 X 11 = 66

6 X 12 = 72

6 X 13 = 78

6 X 14 = 84

6 X 15 = 90

6 X 16 = 96

6 X 17 = 102

6 X 18 = 108

6 X 19 = 114

6 X 20 = 120
```

Snippet-3 Explained

- Humans make mistakes. The programmer missed out on changing the format symbol in the code for `void print(int, int)`, but we don't recommend punishment. He could be overworked, or could only be the maintainer of code someone else wrote years ago! The point is not to blame human flaws (there are many), but to show how code duplication causes errors. This issue arises frequently with overloaded methods, point blank.

- Instead of trying to change human nature, can we change our software? Let's have a closer look at `print()` :
    - The method `void print()` prints the multiplication table for `5` only, in the default range `1` to `10` .
    - The method `void print(int)` outputs the table for any number, but only in the fixed range from `1` to `10` .
    - The method `void print(int,int,int)` displays the table for any number, in any range of entries.

A Solution: Code Reuse

There is something huge we observe in the previous example. All overloaded versions of `print()` have nearly the same code!

- `print(int)` has wider usage potential than `print()` . The latter is a special case, as it prints only the for `5` . We can achieve what `print()` does, by passing a **fixed** parameter value of `5` to `print(int)` . It's simple: invoke `print(int)` within `print()` , by passing `5` to it.
- The point to note is, that **a more specialized function can be implemented-in-terms-of a more general function**.

Let's now reorganize this part of the code.

Snippet-4: Code Reuse

MultiplicationTable.java

```
public static void print() {
    print(5);
}

public static void print(int number) {
    for(int i=1; i<=10;i++) {
        System.out.printf("%d * %d = %d", number, i, number*i).println();
    }
}
```

MultiplicationRunner.java

```
package com.in28minutes.firstjavaproject;

public class MultiplicationRunner {
    public static void main(String[] args) {
        MultiplicationTable table = new MultiplicationTable();
        table.print();
        table.print(8);
    }
}
```

Console Output

*5 * 1 = 5*

*5 * 2 = 10*

*5 * 3 = 15*

*5 * 4 = 20*

*5 * 5 = 25*

*5 * 6 = 30*

*5 * 7 = 35*

*5 * 8 = 40*

*5 * 9 = 45*

*5 * 10 = 50*

*8 * 1 = 8*

*8 * 2 = 16*

*8 * 3 = 24*

*8 * 4 = 32*

*8 * 5 = 40*

*8 * 6 = 48*

*8 * 7 = 56*

*8 * 8 = 64*

*8 * 9 = 72*

*8 * 10 = 80*

Snippet-4 Explained

When we call a method inside another, the method call statement is replaced by its body (with actual arguments replacing the formal ones). In the new definition of `int print()` above, the code executed during the call will be:

```
for(int i=1; i<=10;i++) {
    System.out.printf("%d * %d = %d", 5, i, 5*i).println();
}
```

Extending Code Reuse

- The method `int print(int,int,int)` is a more general version of `int print(int)`. We can achieve what the latter computes, by passing **fixed** range of indexes, namely `1` and `10`, as arguments to the former. have look into he code that follows.

Snippet-5 : Extending code reuse

MultiplicationTable.java

```
public static void print(int number) {
    print(number, 1, 10);
}

public static void print(int number, int from, int to) {
    for(int i=from; i<=to;i++) {
        System.out.printf("%d X %d = %d", number, i, number*i).println();
    }
}
```

MultiplicationRunner.java

```java
package com.in28minutes.firstjavaproject;

public class MultiplicationRunner {
    public static void main(String[] args) {
        MultiplicationTable table = new MultiplicationTable();
        table.print(8);
        table.print(6, 11, 20);
    }
}
```

Console Output

*8 * 1 = 8*

*8 * 2 = 16*

*8 * 3 = 24*

*8 * 4 = 32*

*8 * 5 = 40*

*8 * 6 = 48*

*8 * 7 = 56*

*8 * 8 = 64*

*8 * 9 = 72*

*8 * 10 = 80*

*6 * 11 = 66*

*6 * 12 = 72*

*6 * 13 = 78*

*6 * 14 = 84*

*6 * 15 = 90*

*6 * 16 = 96*

*6 * 17 = 102*

*6 * 18 = 108*

*6 * 19 = 114*

*6 * 20 = 120*

Snippet-5 Explained

- This example merely extended what we did in the previous example. We will will take this extension one level further now! Yes, you guessed right. We will implement `print()` in terms of `print(int,int,int)` .

Snippet-6 : Extending code reuse (contd.)

MultiplicationTable.java

```
    public static void print() {
        print(5, 1, 10);
    }

    public static void print(int number, int from, int to) {
        for(int i=from; i<=to;i++) {
            System.out.printf("%d X %d = %d", number, i, number*i).println();
        }
    }
```

MultiplicationRunner.java

```
    package com.in28minutes.firstjavaproject;

    public class MultiplicationRunner {
        public static void main(String[] args) {
            MultiplicationTable table = new MultiplicationTable();
            table.print();
            table.print(6, 11, 20);
        }
    }
```

Console Output

*5 * 1 = 5*

*5 * 2 = 10*

*5 * 3 = 15*

*5 * 4 = 20*

*5 * 5 = 25*

*5 * 6 = 30*

*5 * 7 = 35*

*5 * 8 = 40*

*5 * 9 = 45*

*5 * 10 = 50*

*6 * 11 = 66*

*6 * 12 = 72*

*6 * 13 = 78*

*6 * 14 = 84*

*6 * 15 = 90*

*6 * 16 = 96*

*6 * 17 = 102*

*6 * 18 = 108*

*6 * 19 = 114*

*6 * 20 = 120*

Snippet-6 Explained

- By extending the same logic of code reuse, the method `int print(int,int,int)` can be used to implement the logic of `int print()` . Just pass in a number parameter `5` , as well as the fixed range parameters `1` and `10` , in a call to the former. `int print()` is thus **implemented-in-terms-of** `int print(int,int,int)` .
- Our new version of `class MultiplicationTable` looks like this:

Snippet-7: Extending Code Reuse (Contd.)

MultiplicationTable.java

```
package com.in28minutes.firstjavaproject;

public class MultiplicationTable {
    public static void print() {
        print(5, 1, 10);
    }

    public static void print(int number) {
        print(number, 1, 10);
    }

    public static void print(int number, int from, int to) {
        for(int i=from; i<=to;i++) {
            System.out.printf("%d X %d = %d", number, i, number*i).println();
        }
    }
}
```

- The logic of the `class MutliplicationRunner` does not change at all:

MultiplicationRunner.java:

```
package com.in28minutes.firstjavaproject;

public class MultiplicationRunner {
    public static void main(String[] args) {
        MultiplicationTable table = new MultiplicationTable();
        table.print();
        table.print(8);
        table.print(6, 11, 20);
    }
}
```

Console Output

*5 * 1 = 5*

*5 * 2 = 10*

*5 * 3 = 15*

*5 * 4 = 20*

*5 * 5 = 25*

*5 * 6 = 30*

*5 * 7 = 35*

*5 * 8 = 40*

*5 * 9 = 45*

*5 * 10 = 50*

*8 * 1 = 8*

*8 * 2 = 16*

*8 * 3 = 24*

*8 * 4 = 32*

*8 * 5 = 40*

*8 * 6 = 48*

*8 * 7 = 56*

*8 * 8 = 64*

*8 * 9 = 72*

*8 * 10 = 80*

*6 * 11 = 66*

*6 * 12 = 72*

*6 * 13 = 78*

*6 * 14 = 84*

*6 * 15 = 90*

*6 * 16 = 96*

*6 * 17 = 102*

*6 * 18 = 108*

*6 * 19 = 114*

*6 * 20 = 120*

Snippet-7 Explained

Neat, isn't it! To make our program school kid friendly, we just need to change one character in the code, take a peek below.

Snippet-8 : Extending Code Reuse (Contd.)

MultiplicationTable.java:

```
package com.in28minutes.firstjavaproject;

public class MultiplicationTable {
    public static void print() {
        print(5, 1, 10);
    }
```

```
        public static void print(int number) {
            print(number, 1, 10);
        }

        public static void print(int number, int from, int to) {
            for(int i=from; i<=to;i++) {
                System.out.printf("%d X %d = %d", number, i, number*i).println();
            }
        }
    }
```

Console Output

5 X 1 = 5

5 X 2 = 10

5 X 3 = 15

5 X 4 = 20

5 X 5 = 25

5 X 6 = 30

5 X 7 = 35

5 X 8 = 40

5 X 9 = 45

5 X 10 = 50

8 X 1 = 8

8 X 2 = 16

8 X 3 = 24

8 X 4 = 32

8 X 5 = 40

8 X 6 = 48

8 X 7 = 56

8 X 8 = 64

8 X 9 = 72

8 X 10 = 80

6 X 11 = 66

6 X 12 = 72

6 X 13 = 78

6 X 14 = 84

6 X 15 = 90

6 X 16 = 96

6 X 17 = 102

6 X 18 = 108

6 X 19 = 114

6 X 20 = 120

Snippet-8 Explained

Software Development is an iterative process. The best code that we want to write does not happen at one go. It starts off at a certain level, and can always be improved. More importantly, such improvements needs to be remembered, to be applied again at different points in the same program, and across programs. This process is called **Code Refactoring**. Thought we'd keep you posted.

Summary

In this step, we:

- Explored how to reorganize the code for *PMT-Challenge* into a `class`
- Understood that overloading works the same way for `class` methods
- Observed that code reuse is possible across overloaded versions of a `class` method

Object Oriented Progamming (OOP)

How do you design great Object Oriented Programs?

Let's find out

Recommended Videos:

- Object Oriented Progamming - Part 1 - https://www.youtube.com/watch?v=NOD802rMMCw
- Object Oriented Progamming - Part 2 - https://www.youtube.com/watch?v=i6EztA-F8UI

Step 01: Object Oriented Progamming (OOP) - Basic Terminology

Let's consider a few examples before we get to Object Oriented Progamming.

Humans think in a step by step process.

Let's say I've to take a flight from London to New York. This is how I would think:

- Take a cab to London Airport
- Check in
- Pass Security
- Board the flight
- Wish the Hostess
- Take Off
- Cruise
- Land
- Get off the plane
- Take a cab to ..

Procedural programming is just a reflection of this thought process. A procedural program for above process would look something like this:

```
takeACabToLondonAirport();
checkIn();
passSecurity();
boardPlane();
wishHostess();
takeOff();
cruiseMode();
land();
getOffPlane();
//...
```

Object Oriented Programming (OOP) brings in a new thought process around this.

How about thinking in terms of the different Actors? How about storing data related to each actor right beside itself? How about giving them some responsiblity and let them do their own actions?

Here's how our program would look like when we think in terms of different actors and give them data and responsibilities

```
    Person
        name
        boardFlight(Plane flight), wishHostess (Hostess hostess), getOffFlight(Plane flight)

    AirPlane
        altitude, pilot, speed, flightMode
        takeOff(), cruiseMode(), land()

    Hostess
        welcome()
```

Do not worry about the implementation details. Focus on the difference in approaches.

We have **encapsulated** data and methods into these entities, which are now called **objects**. We have defined object boundaries, and what it can (and cannot) do.

An object has

- **State** : Its data
- **Behavior** : Its operations

The `position` of an `Airplane` can change over time. The operations that can be performed on an `Airplane` include `takeOff()` , `land()` and `cruiseMode()` . Each of these actions can change its `position` . Therefore, an object's behavior can affects its own state.

It's now time to introduce you to some core **OOP** terms, which will make our future discussions easier.

OOP Terminology

Let's visit and enhance the `Planet` example we had written a few sections ago. This time, let's also explore the conceptual angle.

Planet

```
    class Planet
        name, location, distanceFromSun // data / state / fields
        rotate(), revolve() // actions / behavior / methods

    earth : new Planet
    venus : new Planet
```

Let's look at some **OOP** terminology.

A **class** is a template. An **object** is an instance of a class. In above example, `Planet` is a class. `earth` and `venus` are objects.

- `name`, `location` and `distanceFromSun` compose object state.
- `rotate()` and `revolve()` define object's behavior.

Fields are the elements that make up the object state. Object behavior is implemented through **Methods**.

Each Planet has its own state:

- `name` : "Earth", "Venus"
- `location` : Each has its own orbit
- `distanceFromSun` : They are at unique, different distances from the sun

Each has its own unique behavior:

- `rotate()` : They rotate at different rates (and in fact, different directions!)
- `revolve()` : They revolve round the sun in different orbits, at different speeds

Summary

In this step, we:

- Understood how OOP is different from Prodedural Programming
- Learned about a few basic OOP terms

Step 02: Programming Exercise PE-01

Exercises

In each of the following systems, identify the basic entities involved, and organize them using object oriented terminology:

1. Online Shopping System
2. Person

Solution-1: Online Shopping System

```
Customer
    name, address
    login(), logout(), selectProduct(Product)
```

```
ShoppingCart
    items
    addItem(), removeItem()
```

```
Product
    name, price, quantityAvailable
    order(), changePrice()
```

Solution-2: Person

```
Person
    name, address, hobbies, work
    walk(), run(), sleep(), eat(), drink()
```

Step 03: Creating `MotorBike` `class`

In this series of examples, we want to model your pet mode of transport, a motorbike. We want to create motorbike objects and play around with them.

We will start with two java files:

- ***MotorBike.java***, which contains the `MotorBike` `class` definition. This `class` will encapsulate our motorbike state and behavior
- ***MotorBikeRunner.java***, with `class MotorBikeRunner` holding a `main` method, our program's entry point

Snippet-1: MotorBike Class

MotorBike.java

```java
package com.in28minutes.oops;
    public class MotorBike {
    //behavior
    void start() {
        System.out.println("Bike started!");
    }
}
```

MotorBikeRunner.java

```java
package com.in28minutes.oops;

public class MotorBikeRunner {
    public static void main(String[] args) {
        MotorBike ducati = new MotorBike();
        MotorBike honda = new MotorBike();
        ducati.start();
        honda.start();
    }
}
```

Console Output

Bike started!

Bike started!

Snippet-1 Explained

We started off creating a simple `MotorBike` class with a `start` method. We created a couple of instances and invoked the `start` method on them.

We created two classes because we believe in `Seperation of Concerns` :

- `MotorBike` class is responsible for all its data and behavior.
- `MotorBikeRunner` class is responsible for running MotorBike examples.

Summary

In this step, we:

- Defined a `MotorBike class` allowing us to further explore *OOP* concepts in the next steps

Step 04: Programming Exercise OO-PE-02

Exercises

1. Write a small Java program to create a `Book class`, and then create instances to represent the following book titles:
    - "The Art Of Computer Programming"
    - "Effective Java"
    - "Clean Code"

Solution

Book.java

```java
public class Book {
    private String title;

    public void setTitle(String bookTitle) {
        title = bookTitle;
    }

    public String getTitle() {
        return title;
    }
}
```

BookRunner.java

```java
public class BookRunner {
    public static void main(String[] args) {
        Book taocp = new Book();
        taocp.setTitle("The Art Of Computer Programming");
        Book ej = new Book();
        ej.setTitle("Effective Java");
        Book cc = new Book();
        cc.setTitle("Clean Code");
        System.out.println(taocp.getTitle());
        System.out.println(ej.getTitle());
        System.out.println(cc.getTitle());
    }
}
```

Console Output

The Art Of Computer Programming

Effective Java

Clean Code

Step 05: `MotorBike` - Representing State

An object encapsulates both *state* and *behavior*.

State defines "the condition of the object at a given time". *State* is represented by **member variables**.

In the `MotorBike` example, if we need a `speed` attribute for each `MotorBike` , here is how we would include it.

Snippet-1 : MotorBike with state variable speed

MotorBike.java

```
package com.in28minutes.oops;

public class MotorBike {
    int speed;
    void start() {
        System.out.println("Bike started!");
    }
}
```

MotorBikeRunner.java

```
package com.in28minutes.oops;

public class MotorBikeRunner {
    public static void main(String[] args) {
        MotorBike ducati = new MotorBike();
        MotorBike honda = new MotorBike();
        ducati.start();
        honda.start();

        ducati.speed = 100;
        honda.speed = 80;
        ducati.speed = 20;
        honda.speed = 0;
    }
}
```

Console Output

Bike started!

Bike started!

Snippet-4 Explained

`int speed;` within `MotorBike` , defines a member variable.

It can be accessed within objects such as `ducati` and `honda` , by qualifying it with the object name (`ducati` or `honda`).

```
ducati.speed = 100;
honda.speed = 80;
```

`ducati` has its own value for `speed` , and so does `honda` . These values are independent of each other. Changing one does not affect the other.

Classroom Exercise CE-OO-01

1. Update the `Book` `class` created previously to include a member variable named `noOfCopies` , and demonstrate how it can be set and updated independently for each of the three titles specified earlier.

Solution

TODO

Step 07: `MotorBike` - get() and set() methods

In the previous step. we were merrily modifying the `speed` attribute within the `ducati` and `honda` objects directly, from within the `main()` program.

```
public class MotorBikeRunner {
    public static void main(String[] args) {
        //... Other code
        ducati.speed = 100;
        honda.speed = 0;
    }
}
```

This did work fine, but it breaks the fundamental principle of encapsulation in **OOP**.

"A method of one object, should not be allowed to directly access the state of another object. To do so, such an object should only be allowed to invoke methods on the target object".

In other words, a member variable should not be directly accessible from methods declared outside its `class` .

Snippet-3 : MotorBike class with private attributes

MotorBike.java

```
package com.in28minutes.oops;

public class MotorBike {
    private int speed;

    void start() {
        System.out.println("Bike started!");
    }

    void setSpeed(int speed) {
        this.speed = speed;
    }
}
```

MotorBikeRunner.java

```
package com.in28minutes.oops;

public class MotorBikeRunner {
    public static void main(String[] args) {
        MotorBike ducati = new MotorBike();
        MotorBike honda = new MotorBike();
        ducati.start();
        honda.start();

        ducati.setSpeed(100);
        honda.setSpeed(80);

        ducati.setSpeed(20);
        honda.setSpeed(0);
    }
}
```

Console Output

Bike started!

Bike started!

Snippet-3 Explained

By declaring `speed` as `private`, we provide `MotorBike` with something called **access control**. Java keywords such as `public` and `private` are called **access modifiers**. They control what external objects can access within a given object.

Let's look at `this.speed = speed;` in the body of method `setSpeed()`:

- An member variable always belongs to a specific instance of that `class`.
- A method argument behaves just like a local variable inside that method.
- To differentiate between the two, `this` is used. The expression `this.speed` refers to the member variable `speed` of a `Motorbike` object. `setSpeed()` would be invoked on that very object.

Code written earlier within `MotorBikeRunner`, such as `ducati.speed = 100;` would now result in errors! The correct way to access and modify `speed` is to invoke appropriate methods such as `setSpeed()`, on `MotorBike` objects.

Classroom Exercise CE-OO-02

1. Update the `class Book` to make sure it no longer breaks Encapsulation principles.

Solution

TODO

Summary

In this step, we:

- Learned about the need for access control to implement encapsulation
- Observed that Java provides access modifiers (such as `public` and `private`) for such control
- Understood the need for `get()` and `set()` methods to access object data

Step 08: Accessing Object State

Encapsulation is needed to protect an object's state from direct access by other objects. We were able to protect the state of `MotorBike` objects by declaring `speed` to be `private`. We have created a sort of rigid situation here, since the `private` declaration of `speed` forbids even a *get* access. How do we address this issue? Again, the answer is to provide a method for reading the current `speed`.

Snippet-4 : `getSpeed()` of `MotorBike`

MotorBike.java

```
package com.in28minutes.oops;

public class MotorBike {
    //Same as before

    int getSpeed() {
        return this.speed;
    }
}
```

MotorBikeRunner.java

```java
package com.in28minutes.oops;

public class MotorBikeRunner {
    public static void main(String[] args) {
        MotorBike ducati = new MotorBike();
        MotorBike honda = new MotorBike();
        ducati.start();
        honda.start();

        ducati.setSpeed(100);
        honda.setSpeed(80);
        System.out.printf("Current Ducati Speed is : %d", ducati.getSpeed()).println();
        System.out.printf("Current Honda Speed is : %d", honda.getSpeed()).println();
    }
}
```

Console Output

Bike started!

Bike started!

Current Ducati Speed is : 100

Current Honda Speed is : 80

Snippet-4 Explained

Defining a method such as `getSpeed()` allows us to access the current `speed` of a `MotorBike` object.

```java
    int getSpeed() {
        return this.speed;
    }
```

> Eclipse has a very handy feature. When the state elements (member variables) of a class have been defined, it can generate default get() and set() methods for each of them. You would want to use this regularly, to save on time and typing effort. `Right click on class > Generate Source > Generate Getters and Setters`

Summary

In this step, we:

- Understood how access control forces us to provide `get()` methods as well
- Explored a few Eclispe tips to generate `get()` and `set()` versions for each `class` attribute

Step 10: Default Object State

What happens if a data element inside an object is not initialized with a value?

Snippet-5 : Default Initialization of Object State

MotorBike.java

```java
//Same as before
```

MotorBikeRunner.java

```
package com.in28minutes.oops;

public class MotorBikeRunner {
    public static void main(String[] args) {
        MotorBike honda = new MotorBike();
        System.out.printf("Current Honda Speed is : %d", honda.getSpeed()).println();
    }
}
```

Console Output

Current Honda Speed is : 0

Snippet-6 Explained

When we instantiate an object, all its state elements are always initialized, to the default values of their types. Inside `MotorBike` , `speed` is declared to be an `int` , and so is initialized to `0` . This happens even before any method of `MotorBike` is called, including `start()` .

You can see that `honda.getSpeed()` printed `0` even though we did not explictly initialize it.

Default

Summary

In this step, we:

- Learnt that default values are assigned to object member variables.

Step 10: Encapsulation: Its Advantages

At the heart of encapsulation, is the need to protect object's state. From whom? Other objects.

Let's look at an example to understand Encapsulation better.

Snippet-7 : Advantage of Encapsulation

MotorBike.java

```
package com.in28minutes.oops;

public class MotorBike {
    private int speed;

    public void start() {
        System.out.println("Bike started!");
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

    public int getSpeed() {
        return this.speed;
    }
}
```

MotorBikeRunner.java

```
package com.in28minutes.oops;

public class MotorBikeRunner {

    public static void main(String[] args) {
        MotorBike ducati = new MotorBike();
        ducati.start();
        ducati.setSpeed(-100);
        System.out.printf("Current Ducati Speed is : %d", ducati.getSpeed()).println();
    }
}
```

Console Output

Bike started!

Current Ducati Speed is : -100

Snippet-01 Explained

For a motorbike, `-100` might be an invalid speed. Currently, we do not have any validation. Having a method `setSpeed` allows us to add validation.

Let's see how to do that.

Snippet-02 : Speed Validity Check

MotorBike.java

```
package com.in28minutes.oops;

public class MotorBike {

    //Other code as is

    public void setSpeed(int speed) {
        if(speed > 0)
            this.speed = speed;
    }

}
```

MotorBikeRunner.java

```
//Same as before
```

The output of this snippet is:

Bike started!

Current Ducati Speed is : 0

Snippet-8 Explained

`setSpeed()` checks `if(speed > 0)` and does not update speed for negative values.

> This is not perfect solution because the caller of the `setSpeed` method assumes that he was successful. Ideally, we should throw an Exception indicating validation error. We will talk about Exceptions later.

Summary

In this step, we:

- Explored the first advantage of encapsulation - A provision for adding data validation
- Highlighted how such validation can be done, using the `Motorbike` example

Step 11: Encapsulation - Advantages (Code Reuse)

We've understood quite a few things about encapsulation under *OOP*.

Suppose at different points of time, we want to increase the speeds of both Honda and Ducati bikes by a fixed amount, say `100 mph` . The logic would be simple, right? Fetch the current `speed` of each bike, increment that fetched value by `100` , and then set the new value back into that bike's `speed` . The following example puts the above logic into action.

Snippet-1 : Bulky Speed Increment code

MotorBike.java

```
// No Change
```

MotorBikeRunner.java

```

    package com.in28minutes.oops;

    public class MotorBikeRunner {
        public static void main(String[] args) {
            MotorBike ducati = new MotorBike();
            MotorBike honda = new MotorBike();
            ducati.start();
            honda.start();
            ducati.setSpeed(100);

            System.out.printf("Earlier Ducati Speed is : %d", ducati.getSpeed()).println();
            System.out.printf("Earlier Honda Speed is : %d", honda.getSpeed()).println();

            int ducatiSpeed = ducati.getSpeed();
            ducatiSpeed = ducatiSpeed + 100;
            ducati.setSpeed(ducatiSpeed);

            int hondaSpeed = honda.getSpeed();
            hondaSpeed = hondaSpeed + 100;
            honda.setSpeed(hondaSpeed);

            System.out.printf("Later Ducati Speed is : %d", ducati.getSpeed()).println();
            System.out.printf("Later Honda Speed is : %d", honda.getSpeed()).println();
        }
    }
```

The output of this snippet is:

Bike started!

Bike started!

Earlier Ducati Speed is : 100

Earlier Honda Speed is : 0

Later Ducati Speed is : 200

Later Honda Speed is : 100

Snippet-1 Explained

Notice the repeated code within `MotorBikeRunner` ? The code for updating the `speed` of `ducati` , is almost the same as that for `honda` . Remember at the start of this book, we said something like this:

"The goal of any computer program is to make a task easier, less cumbersome and more elegant for the programmer."

OOP achieves all thus through encapsulation! The idea is to *encapsulate* repeated logic within a method, and pass object-specific information to it as arguments. The next example shows you one way of doing it.

Snippet-2 : Speed Increase through Code Encapsulation

MotorBike.java

```
package com.in28minutes.oops;

public class MotorBike {
    //Other code same as before
    public void increaseSpeed(int howMuch) {
        this.speed = this.speed + howMuch;
    }
}
```

MotorBikeRunner.java

```
package com.in28minutes.oops;

public class MotorBikeRunner {
    public static void main(String[] args) {

        MotorBike duati = new MotorBike();
        MotorBike honda = new MotorBike();
        ducati.start();
        honda.start();

        ducati.setSpeed(100);

        System.out.printf("Earlier Ducati Speed is : %d", ducati.getSpeed()).println();
        System.out.printf("Earlier Honda Speed is : %d", honda.getSpeed()).println();

        ducati.increaseSpeed(100);
        honda.increaseSpeed(100);

        System.out.printf("Later Ducati Speed is : %d", ducati.getSpeed()).println();
        System.out.printf("Later Honda Speed is : %d", honda.getSpeed()).println();
    }
}
```

The output of this snippet is:

Bike started!

Bike started!

Earlier Ducati Speed is : 100

Earlier Honda Speed is : 0

Later Ducati Speed is : 200

Later Honda Speed is : 100

Snippet-2 Explained

The method `increaseSpeed()` has been added to `MotorBike`. It can be invoked on the `ducati` and `honda` objects.

Let's now add a feature to `MotorBike`, by which `speed` can be decreased.

Snippet-3 : Speed Increase And Decrease through Code Encapsulation

MotorBike.java

```java
package com.in28minutes.oops;

public class MotorBike {
    // Other methods same as before
    public void decreaseSpeed(int howMuch) {
        this.speed = this.speed - howMuch;
    }
}
```

MotorBikeRunner.java

```java
package com.in28minutes.oops;

public class MotorBikeRunner {
    public static void main(String[] args) {
        MotorBike ducati = new MotorBike();
        MotorBike honda = new MotorBike();
        ducati.start();
        honda.start();

        ducati.setSpeed(100);
        System.out.printf("Earlier Ducati Speed is : %d", ducati.getSpeed()).println();
        System.out.printf("Earlier Honda Speed is : %d", honda.getSpeed()).println();

        ducati.increaseSpeed(100);
        honda.increaseSpeed(100);
        ducati.decreaseSpeed(50);
        honda.decreaseSpeed(50);

        System.out.printf("Later Ducati Speed is : %d", ducati.getSpeed()).println();
        System.out.printf("Later Honda Speed is : %d", honda.getSpeed()).println();

        ducati.decreaseSpeed(200);
        honda.decreaseSpeed(200);

        System.out.printf("Final Ducati Speed is : %d", ducati.getSpeed()).println();
        System.out.printf("Final Honda Speed is : %d", honda.getSpeed()).println();
    }
}
```

The output of this snippet is:

Bike started!

Bike started!

Earlier Ducati Speed is : 100

Earlier Honda Speed is : 0

Later Ducati Speed is : 150

Later Honda Speed is : 50

Final Ducati Speed is : -50

Final Honda Speed is : -150

Snippet-3 Explained

The method `decreaseSpeed()` has been added to `MotorBike` . It can be invoked on the `ducati` and `honda` objects inside the `main()` method of `MotorBikeRunner` .

On of the things you can observe again is ***Negative Speed Values***. Our validation needs to be improved.

Snippet-4 : Validation across methods, repeated!

MotorBike.java

```
package com.in28minutes.oops;

public class MotorBike {

    //Same as before

    public void increaseSpeed(int howMuch) {
        if(this.speed + howMuch > 0)
           this.speed = this.speed + howMuch;
    }

    public void decreaseSpeed(int howMuch) {
        if(this.speed - howMuch > 0)
           this.speed = this.speed - howMuch;
    }
}
```

MotorBikeRunner.java

```
//Same as before
```

The output of this snippet is:

Bike started!

Bike started!

Earlier Ducati Speed is : 100

Earlier Honda Speed is : 0

Later Ducati Speed is : 150

Later Honda Speed is : 50

Final Ducati Speed is : 150

Final Honda Speed is : 50

Snippet-4 Explained

We have achieved data validation, because attempts to decrease the `speed` of `ducati` and `honda` below `0` are now ignored.

But this has come at a cost, which is *code bloat*.

How do we reduce duplication?

Snippet-5: Validation by code reuse

MotorBike.java

```java
package com.in28minutes.oops;

public class MotorBike {
    //Other methods same as before


    public void setSpeed(int speed) {
        if(speed > 0)
            this.speed = speed;
    }

    public void increaseSpeed(int howMuch) {
        setSpeed(this.speed + howMuch);
    }

    public void decreaseSpeed(int howMuch) {
        setSpeed(this.speed - howMuch);
    }
}
```

MotorBikeRunner.java

```java
//Same as before
```

The output of this snippet is:

Bike started!

Bike started!

Earlier Ducati Speed is : 100

Earlier Honda Speed is : 0

Later Ducati Speed is : 150

Later Honda Speed is : 50

Final Ducati Speed is : 150

Final Honda Speed is : 50

Snippet-5 Explained

The idea behind this solution is, that an *update* is the same as a `set()` operation. Since `setSpeed()` already has a validation check, it can be called inside both `increaseSpeed()` and `decreaseSpeed()` with appropriate parameters.

In this way, the validation logic would be reused across update methods.

> Be always careful. Duplication of logic makes your code difficult to maintain.

Summary

In this step, we:

- Started exploring the next advantage of encapsulation - code reuse
- Mapped this understanding to the `MotorBike` example, building on data validation

Step 12: Programming Exercise PE-OO-03

Exercises

1. Use an encapsulation technique to write methods for the `Book class`, that
    - Increase the number of books
    - Decrease the number of books

Solution

BookRunner.java

```
    public class BookRunner {
        public static void main(String[] args) {
            Book taocp = new Book();
            taocp.setTitle("The Art Of Computer Programming");
            Book ej = new Book();
            ej.setTitle("Effective Java");
            Book cc = new Book();
            cc.setTitle("Clean Code");

            System.out.println(taocp.getTitle());
            System.out.println(ej.getTitle());
            System.out.println(cc.getTitle());

            taocp.increaseCopies(10);
            ej.increaseCopies(15);
            cc.increaseCopies(20);
            taocp.decreaseCopies(5);
            ej.decreaseCopies(10);
            cc.decreaseCopies(15);

            System.out.println(taocp.getNumberOfCopies());
            System.out.println(ej.getNumberOfCopies());
            System.out.println(cc.getNumberOfCopies());
        }
    }
```

Book.java

```
    public class Book {
        private String title;
        private int numberOfCopies;

        public void setTitle(String bookTitle) {
            title = bookTitle;
        }

        public String getTitle() {
            return title;
        }

        public void setNumberOfCopies(int numberOfCopies) {
            if(numberOfCopies > 0) {
                this.numberOfCopies = numberOfCopies;
            }
        }

        public int getNumberOfCopies() {
            return numberOfCopies;
        }

        public void increaseCopies(int howMuch) {
            setNumberOfCopies(numberOfCopies + howMuch);
        }

        public void decreaseCopies(int howMuch) {
            setNumberOfCopies(numberOfCopies - howMuch);
        }
    }
```

Console Output

The Art Of Computer Programming

Effective Java

Clean Code

5

5

5

Step 13: Introducing Constructors

When we create Ducati and Honda motorbikes, we may want to configure them with some start speeds.

Suppose our whim is that a Ducati bike starts with 100 mph, and a Honda with 200 mph.

Snippet-1: MotorBike Constructor

MotorBike.java

```
    package com.in28minutes.oops;

    public class MotorBike {
        private int speed;


        MotorBike(int speed) {
            if(speed > 0)
                this.speed = speed;
        }

        //Same as before

    }
```

MotorBikeRunner.java

```
    package com.in28minutes.oops;

    public class MotorBikeRunner {
        public static void main(String[] args) {
            MotorBike ducati = new MotorBike(100);
            MotorBike honda = new MotorBike(200);
            ducati.start();
            honda.start();
            System.out.printf("Earlier Ducati Speed is : %d", ducati.getSpeed()).println();
            System.out.printf("Earlier Honda Speed is : %d", honda.getSpeed()).println();
        }
    }
```

The output of this snippet is:

Bike started!

Bike started!

Earlier Ducati Speed is : 100

Earlier Honda Speed is : 200

Snippet-1 Explained

We defined a single-argument constructor for `MotorBike` , whosw definition looks like this:

`public MotorBike(int speed){ /* Constructor Code Goes Here */ }`

The constructor is a method, whose name is the same as the `class` name. All Java rules for a method apply to constructors as well. Constructor cannot be directly called.

A constructor is always invoked when a `class` object is created, using the `new` keyword. A constructor for a `class` could accept zero, one or more than one arguments. Let's next write some full-blooded code for a `MotorBike` constructor.

Summary

In this step, we were introduced to the concept of a `class` constructor

Programming Exercise PE-OO-04, And More On Constructors

Exercises

1. Rewrite the `Book` `class` solution by using a constructor, which accepts an integer argument specifying the initial number of copies to be published:
   - "The Art Of Computer Programming" : 100 copies
   - "Effective Java" : 75 copies
   - "Clean Code" : 60 copies

Solution To PE-OO-04

BookRunner.java

```
public class BookRunner {
    public static void main(String[] args) {
        Book taocp = new Book(100);
        taocp.setTitle("The Art Of Computer Programming");

        Book ej = new Book(75);
        ej.setTitle("Effective Java");

        Book cc = new Book(60);
        cc.setTitle("Clean Code");

        System.out.println(taocp.getTitle());
```

```
            System.out.println(taocp.getNumberOfCopies());

            System.out.println(ej.getTitle());`
            System.out.println(ej.getNumberOfCopies());

            System.out.println(cc.getTitle());
            System.out.println(cc.getNumberOfCopies());
        }
    }
```

Book.java

```
    public class Book {
        private String title;
        private int numberOfCopies;

        public Book(int numberOfCopies) {
            this.numberOfCopies = numberOfCopies;
        }

        public void setTitle(String bookTitle) {
            title = bookTitle;
        }

        public String getTitle() {
            return title;
        }

        public int getNumberOfCopies() {
            return numberOfCopies;
        }
    }
```

Console Output

The Art Of Computer Programming

100

Effective Java

75

Clean Code

60

More on Constructors

We enjoyed defining the `MotorBike` `class` and checking out the behavior of its instances, `ducati` and `honda` .

When we started off , we created instances of `MotorBike` classes using:

```
MotorBike ducati = new MotorBike();
```

Did you notice something familiar? Doesn't the expression `new MotorBike()` look like a constructor call?

As it turns out, it is a constructor call on `MotorBike` !

When we define a `class` in Java (even a seemingly empty one), some behind-the-scene magic happens. Consider one such `class` `Cart` :

```
class Cart {

};
```

This `class` has neither state, nor behavior. When we try to create instances of this "empty" `class` :

```
class CartRunner {
    public static void main(String[] args) {
        Cart cart = new Cart();
    }
}
```

The code seems to compile, execute and get initialized quite smoothly! What happens is, that the compiler silently generates a **default constructor** for `Cart` .

A default constructor is one that accepts no arguments. It is as if the `Cart` `class` were defined this way:

```
class Cart {
    public Cart() {
    }
};
```

A constructor may also have overloaded definitions.

Let's now try to create `MotorBike` instances with default initialization, just as we did with `Cart` .

Snippet-2 : Default Motorbike Construction?

MotorBike.java

```
package com.in28minutes.oops;

public class MotorBike {

    private int speed;

    MotorBike(int speed) {
        if(speed > 0)
            this.speed = speed;
    }
}
```

MotorBikeRunner.java

```
package com.in28minutes.oops;

public class MotorBikeRunner {
    public static void main(String[] args) {
        MotorBike ducati = new MotorBike(100);
        MotorBike honda = new MotorBike(200);
        MotorBike yamaha = new MotorBike();
    }
}
```

Console Output

Compiler Error

Snippet-2 Explained

The compiler flags an error with ***MotorBikeRunner.java***! `MotorBike yamaha = new MotorBike();` is failing compilation. Why?

No default constructor is generated here! If you provide any constructor definition in a class, the compiler will not add a default constructor. **Do them all yourself, if you don't like what I do for you!** is what it yells back.

If you need the default constructor, you can explicitly add it.

Snippet-3 : Programmer-defined default constructor

MotorBike.java

```
package com.in28minutes.oops;

public class MotorBike {
    //state
    private int speed;

    //behavior
    MotorBike() {
        this(5);
    }

    MotorBike(int speed) {
        if(speed > 0)
            this.speed = speed;
    }

}
```

MotorBikeRunner.java

```
package com.in28minutes.oops;

public class MotorBikeRunner {
    public static void main(String[] args) {
        MotorBike ducati = new MotorBike(100);
        MotorBike honda = new MotorBike(200);
        MotorBike yamaha = new MotorBike();
        System.out.printf("Earlier Yamaha Speed is : %d", yamaha.getSpeed()).println();
    }
}
```

The output of this snippet is:

Earlier Yamaha Speed is : 5

Snippet-3 Explained

We defined a zero-argument constructor `MotorBike()` , to enable default object initialization. But what are we doing in its body with the statement `this(5);` ?

We have called the constructor `MotorBike(int)` , qualified with the `this` keyword, within `Motorbike()` .

Primitive Data Types

Earlier, we looked at basic Java types (including `int` , `double` and `boolean`), and got a little familiar with them.

We used literal values, declared variables and formed expressions using them.

Java **primitive types** include:

- Integer Types
  - `byte`
  - `short`
  - `int`
  - `long`
- Floating-Point Types
  - `float`
  - `double`
- Character Type
  - `char`
- Logical Type
  - `boolean`

In this section, let's play with each of these types to understand them further.

Step 01: The Integer Types

Integers are not much of a mystery, are they? They've been part of us since our school days, and we normally prefer them over other numbers (they scare us less!).

Java supports them with ease, and we have coded quite a few examples using them, already.

Java also has a **wrapper class** corresponding to each of them, which make their primitive versions more versatile. The wrapper classes we are talking about are:

- `Byte` : for `byte`
- `Short` : matching `short`
- `Integer` corresponding to `int`
- `Long` : about `long`

Let's see how we can work with them.

Snippet-01 : Integer Sizes

Look at all the information these tiny classes hold for you! As they say, ***"fore-warned is fore-armed"***. Depending on the data (range and size) your program handles, you decide which types to use, to store them.

```
    jshell> Byte.SIZE
    $1 ==> 8
    jshell> Byte.BYTES
    $2 ==> 1
    jshell> Byte.MAX_VALUE
    $3 ==> 127
    jshell> Byte.MIN_VALUE
    $4 ==> -128
    jshell> Short.BYTES
    $5 ==> 2
    jshell> Integer.BYTES
    $6 ==> 4
    jshell> Long.BYTES
    $7 ==> 8
    jshell> Short.MAX_VALUE
```

```
        $8 ==> 32767
        jshell> Integer.MAX_VALUE
        $8 ==> 2147483647
        jshell> int i = 100000;
        i ==> 100000
        jshell> long l = 50000000000;
        | Error:
        | integer number too large: 50000000000
        | long l = 50000000000;
        |_________^
        jshell> long l = 50000000000l;
        l ==> 50000000000
        jshell>
```

Integer Type Conversions

Problems will (and should) arise if we attempt to store large data into smaller bins. The compiler warns the programmer about such issues by flagging errors. However, if the programmer is aware of the risks and intends to go ahead, **explicit casts** are her tools to push the code through.

Operations in the other direction (storing a smaller data value in a larger bin), is a piece of cake for the compiler. Such a conversion is called an **implicit cast**.

Snippet-02 : Integer Type Conversions

Attempting to store a `long` value into an `int` variable will give us a compiler error. However, you can use an explicit cast, such as `i = (int) l;` .

```
        jshell> int i = 100000;
        i ==> 100000
        jshell> long l = 50000000000l;
        l ==> 50000000000
        jshell> i = l;
        | Error :
        | incompatible types: possible lossy conversion from long to int
        | i = l;
        |_____^
        jshell> i = (int) l;
        i ==> -1539607552
        jshell> l = i;
        l ==> -1539607552
        jshell>
```

The compiler is not responsible for the type-safety of this statement. The onus is on me, the programmer.

Remember our earlier statement on possible incorrect program behavior? As you can see, a different value got stored in `i` .

Summary

In this step, we:

- Explored the wrapper classes present for the primitive integer types
- Understood the different capacities and data ranges of these types
- Examined how to use explicit and implicit casts

Step 02: Integer Representations, And Other Puzzles

In a decimal system, the allowed digits are `0` through `9` . When a value of `10` is encountered, the number of digits increases by 1, and its representation is " `10` ".

Those familiar with number systems would know that decimal is not the only system that computers understand.

Other number systems that the Java language supports are **Octal** (Base-8) and **Hexadecimal** (Base-16).

In an Octal system, the allowed digits are `0` through `7` , and a value `8` is represented by " `010` ". The leading `0` is added only to distinguish the octal format from the decimal one.

In a Hexadecimal system, the allowed digits are `0` through `9` , followed by `a` through `f` (or `A` through `F`). A value of `16` is represented by " `0x10` ". Here, a leading `0x` is added to help the compiler recognize Hexa decimal representation.

Let's see how Java supports these three number systems.

Snippet-01 : Storing Octal and Hexadecimal in Integer types

There are no number-system specific integer types in Java! The same `int` type is used to store decimal, octal and hexadecimal values.

If we adhere to to number system conventions about valid digits and understand the compiler hints, we get no surprises.

```
jshell> int eight = 010;
eight ==> 8
jshell> int sixteen = 0x10;
sixteen ==> 16
jshell> int fifteen = 0xf;
fifteen ==> 15
jshell> int eight = 08;//8 is invalid octal digit
| Error:
| integer number too large : 08
| int eight = 08;
|____________^
jshell> int big = 0xbbaacc;
big ==> 12298956
jshell>
```

Snippet-02 : More Integer Type-casting

There are two kinds of assignments:

- Literal-to-variable assignment: With `short s = 123456;` , the data is clearly out of range (this is known at compile-time). The compiler flags an error.
- Variable-to-variable assignment: Consider `sh = in;` . The value stored in `int in` at that stage was `4567` , which is well within the range of the `short` type. The compiler chooses not to take chances and flags an error. This can again be preempted with a explicit cast `sh = (short) in` .

```
jshell> byte b = 128;
| Error:
| incompatible types: possible lossy conversion from int to byte
| byte b = 128;
|__________^__^
jshell> short s = 123456;
| Error:
| incompatible types: possible lossy conversion from int to short
| short s = 123456;
|___________^______^
jshell> short sh = 3456;
sh ==> 3456
jshell> int in = 4567;
in ==> 3456
jshell> sh = in;
```

```
    | Error:
    | incompatible types: possible lossy conversion from int to short
    | sh = in;
    |______^
    jshell> sh = (short) in;
    sh ==> 4567
    jshell> int num = sh;
    num ==> 4567
    jshell>
```

Built-In Operators For Integer Types

We already had a glimpse of arithmetic operators for the integer types:

- `+`
- `-`
- `*`
- `/`
- `%`
- `++` (both prefix and post-fix increment)
- `--` (both prefix and post-fix increment)

The increment and decrement operators are an interesting case, as they are actually short-hands for multiple statements. When we use their prefix and post-fix versions, we need to look out for side-effects.

Snippet-03 : Increment & Decrement Operators

With post-fix increment, such as in `int j = i++;` , the increment takes place *after* the assignment is done. `j` gets the value before increment.

```
    jshell> int i = 10;
    i ==> 10
    jshell> int j = i++;
    j ==> 10
    jshell> i
    i ==> 11
```

When prefix increment is involved, as with `int n = ++m;` , the increment is carried out *before* the assignment. `n` gets the value after increment.

```
    jshell> int m = 10;
    m ==> 10
    jshell> int n = ++m;
    n ==> 11
    jshell> m
    m ==> 11
```

With post-fix decrement, as with `int l = k--;` , the decrement occurs *after* the assignment is done. As far as prefix decrement is concerned, such as in `int q = --p;` , the decrement is performed *before* the assignment.

```
    jshell> int k = 10;
    k ==> 10
    jshell> int l = k--;
    l ==> 10
    jshell> k
    k ==> 9
    jshell> int p = 10;
    p ==> 10
```

```
jshell> int q = --p;
q ==> 9
jshell> p
p ==> 9
jshell>
```

Summary:

In this step, we:

- Looked at the number-systems supported in Java for integers
- Examined how prefix and post-fix versions work for increment and decrement operators

Step 03: Classroom Exercise CE-01 (With Solutions)

Exercise Set

1. Create a Java `class BiNumber` that stores a pair of integers, and has the following functionality:
   - Can be created by passing its initial two numbers to store
   - Must Support Addition and Multiplication operations on the stored integers
   - An operation to double the values of both numbers
   - Operations to access each number individually

In short, we must be able to write code like this in the `main` method of our runner class:

```
BiNumber numbers = new BiNumber(2, 3);
System.out.println(numbers.add());
System.out.println(numbers.multiply());
numbers.double();
System.out.println(numbers.getNumber1());
System.out.println(numbers.getNumber2());
```

Solution to CE-01

BiNumber.java

```
package com.in28minutes.primitive.datatypes;

public class BiNumber {
    private int number1;
    private int number2;

    public BiNumber(int number1, int number2) {
        this.number1 = number1;
        this.number2 = number2;
    }

    public int add() {
        return number1 + number2;
    }

    public int multiply() {
        return number1 * number2;
    }

    public void doubleValue() {
        number1 *= 2;
```

```
            number2 *= 2;
        }

        public int getNumber1() {`
            return number1;
        }

        public int getNumber2() {
            return number2;
        }

        public void setNumber1(int number1) {
            this.number1 = number1;
        }

        public void setNumber2(int number2) {
            this.number2 = number2;
        }
    }
```

BiNumberRunner.java

```
    package com.in28minutes.primitive.datatypes;

    public class BiNumberRunner {
        public static void main(String[] args) {
            BiNumber numbers = new BiNumber(2, 3);
            System.out.println(numbers.add());
            System.out.println(numbers.multiply());
            numbers.doubleValue();
            System.out.println(numbers.getNumber1());
            System.out.println(numbers.getNumber2());
        }
    }
```

Console Output

5

6

4

6

Step 05: Floating-Point Types

You would recall there are two types in Java to support floating-point numbers:

- `double` : The default type for floating-point literals, with a capacity of 8 bytes
- `float` : A narrower, less precise representation for floating-point data. It has a capacity of 4 bytes.

Let's quickly refresh what we know, with a few code snippets.

Snippet-01 : double and float

Default floating point type in Java is `double` . A `float` literal must be accompanied with a trailing `f` or `F` .

```
    jshell> float f = 34.5;
    | Error:
    | incomaptible types: possible lossy conversion from double to float
```

```
| float f = 34.5;
|_________^___^
jshell> float f = 34.5f;
f ==> 34.5
jshell> float fl = 34.5F;
fl ==> 34.5
jshell> double d = 34.5678;
d ==> 34.5678
jshell> float flo = d;
| Error:
| incomaptible types: possible lossy conversion from double to float
| float flo = d;
|__________^
jshell> float flo = (float) d;
flo ==> 34.567
jshell>
```

Snippet-02 : Operators for type double

You can use operators `++` , `--` and `%` on double.

```
jshell> double dbl = 34.5678;
dbl ==> 34.5678

jshell> dbl++
$3 ==> 34.5678

jshell> dbl
dbl ==> 35.5678

jshell> dbl--
dbl ==> 35.5678
jshell> dbl % 5
dbl ==> 4.567799999999998
```

You would need an explicit cast to convert a float to an integer value `int i = (int)f` .

```
jshell> float f = 34.5678f;
f ==> 34.5678
jshell> int i = f;
| Error:
| incomaptible types: possible lossy conversion from float to int
| int i = f;
|_______^
jshell> int i = (int)f;
i ==> 34
jshell> float fl = i;
fl ==> 34.0
jshell>
```

Summary

In this step, we:

- Saw how we create literals and variables of the floating-point types
- Understood the differences between `double` and `float`

Step 06: Introducing BigDecimal

Compact though they are, `double` and `float` are not very precise representations of floating-point numbers.

In fact, they are not used in computations that require high degrees for accuracy, such as scientific experiments and financial applications. The next example shows you why.

```
jshell> 34.56789876 + 34.2234
$1 ==> 68.79129875999999
```

The literal expression `34.56789876 + 34.2234` should evaluate to `68.79129876`. Above addition is not really accurate.

This is due to a flaw in floating-point representations, used in the `double` and `float` types.

The `BigDecimal` class was introduced in Java to tide over these problems.

Accuracy of `BigDecimal` representation is retained only when `String` literals are used to build it.

```
jshell> BigDecimal number1 = new BigDecimal("34.56789876");
number1 ==> 34.56789876
jshell> BigDecimal number2 = new BigDecimal("34.2234");
number2 ==> 34.2234_
```

A `BigDecimal` type can be used to create only **immutable** objects. The values in an *immutable* object cannot be changed after creation.

You can see that the value of number1 is not changed on executing `number1.add(number2)`. To get the result of the sum, we create a new variable `sum`.

```
jshell> number1.add(number2);
$2 ==> 68.79129876
jshell> number1
number1 ==> 34.56789876
jshell> BigDecimal sum = number1.add(number2);
sum ==> 68.79129876
```

Summary

In this step, we:

- Learned that `double` and `float` are not very precise data types
- Were introduced to the `BigDecimal` built-in data type
- Understood that `BigDecimal` is immutable
- Saw that accuracy is best achieved when you construct `BigDecimal` data using string literals

Step 06: BigDecimal Operations

Let's look at a few other operations defined in the `BigDecimal` class.

Snippet-01: Arithmetic Operations

All BigDecimal operations support only BigDecimal operands.

```
jshell> BigDecimal number1 = new BigDecimal("11.5");
number1 ==> 34.56789876
jshell> BigDecimal number2 = new BigDecimal("23.45678");
number2 ==> 23.45678
jshell> number1.add(number2);
$1 ==> 34.95678
```

`BigDecimal` does not jell well with primitive types.

```
jshell> int i = 5;
i ==> 5
jshell> number1.add(i);
| Error:
| incompatible types: int cannot be converted to java.math.BigDecimal
| number1.add(i);
|_____________^
```

We can add, multiple, divide or subtract `BigDecimal` values.

```
jshell> number1.add(new BigDecimal(i));
$2 ==> 16.5
jshell> number1.multiply(new BigDecimal(i));
$3 ==> 67.5
jshell> number1.divide(new BigDecimal(100));
$4 ==> 0.115
jshell>
```

Summary

In this step, we:

- Explored the `BigDecimal` methods for doing some basic arithmetic
- Found that `BigDecimal` has constructors accepting most basic Java numeric types

Step 07: Classroom Exercise CE-02

Exercise-Set

Write a Program that does a Simple Interest computation for a Principal amount. Recall that the formula for such a calculation is:

`Total amount (TA) = Principal Amount (PA) + ( PA * Simple Interest (SI) Rate * Duration In Years (N))`

In essence, write a `SimpleInterestCalculator` `class` that can be used in the following manner:

```
SimpleInterestCalculator calculator = new SimpleInterestCalculator("4500.00", "7.5");
BigDecimal totalValue = calculator.calculateTotalValue(5); //5 year duration
System.out.println(totalValue);
```

Solution To CE-02

SimpleInterestCalculatorRunner.java

```
package com.in28minutes.primitive.datatypes;
import java.math.BigDecimal;

public class SimpleInterestCalculatorRunner {
    public static void main(String[] args) {
        SimpleInterestCalculator calculator = new SimpleInterestCalculator("4500.00", 7.5");
        BigDecimal totalValue = calculator.calculateTotalValue(5); //5 year duration
        System.out.println(totalValue);
    }
}
```

SimpleInterestCalculator.java

```
    package com.in28minutes.primitive.datatypes;
    import java.math.BigDecimal;

    public class SimpleInterestCalculatorRunner {
        BigDecimal principal;
        BigDecimal interest;

        public SimpleInterestCalculator(String principal, String interest) {
            this.principal = new BigDecimal(principal);
            this.interest = new BigDecimal(interest).divide(new BigDecimal(100));
        }

        public BigDecimal calculateTotalValue(int noOfYears)
            //Total Value = Principal  + Principal * Interest* Years
            BigDecimal totalValue = prinipal.add(principal.multiply(interest).multiply(new BigDecima
            return totalValue;`
        }
}
```

Console Output:

6187.50000

Tip: The `import` Statement

The Java `import` statement is Required in each source file that uses a `class` from another `package` . Hence, both ***SimpleInterestCalculator.java*** (`class` definition) and ***SimpleInterestCalculatorRunner.java*** (runner `class` definition) need to import `class` `java.math.BigDecimal` .

`package java.lang.*` is imported by default.

The `.*` suffix indicates that all classes in the `package` `java.lang` are being imported.

Summary

In this step, we:

- Used `BigDecimal` values in a stand-alone Java program
- Learnt how to make use of built-in Java packages, through the `import` statement

Step 08: `boolean` , Relational and Logical Operators

The Java `boolean` data type is one that holds only one of two values: `true` or `false` . Both labels are case-sensitive. We have seen examples of built-in comparison operators, such as `==` , `!=` and `>` , that evaluate expressions to give us `boolean` results. Let us do a quick recap of some of these.

Snippet-01 : Relational Operators : Recap

Relational Operators are used mainly for comparison. They accept operands of non- `boolean` primitive data types, and return a `boolean` value.

```
    jshell> int i = 7;
    i ==> 7
    jshell> i > 7;
    $1 ==> false
    jshell> i >= 7;
    $2 ==> true
```

```
jshell> i < 7;
$3 ==> false
jshell> i <= 7;
$4 ==> true
jshell> i == 7;
$5 ==> true
jshell> i == 8;
$6 ==> false
jshell>
```

Logical Operators

Java also has logical operators that are used in expressions. Logical operators expect `boolean` operands. Expressions involving these are used for forming `boolean` conditions in code, including within `if` , `for` and `while` statements. The prominent logical operators are:

- `&&` : logical **AND**
- `||` : **OR**
- `!` : **NOT**

Let's learn how we use them in our code.

Snippet-02 : Logical Operators

We would want to find if `i` is between `15` and `25` . An expression `i >= 15 && i <= 25` can be used. Details below.

```
jshell> int i = 17;
i ==> 17
jshell> i >= 15
$1 ==> true
jshell> i <= 25
$2 ==> true
jshell> i >= 15 && i <= 25
$3 ==> true
jshell> i == 30;
i ==> 30
jshell> i >= 15 && i <= 25
$4 ==> false
jshell> i == 5;
i ==> 30
jshell> i >= 15 && i <= 25
$5 ==> false
```

An expression with `&&` evaluates to `true` only if **both** its `boolean` operands evaluate to `true` .

```
jshell> true && true
$6 ==> true
jshell> true && false
$7 ==> false
jshell> false && true
$8 ==> false
jshell> false && false
$9 ==> false
jshell>
```

Snippet-02 Explained

The next example helps us visualize truth tables for the prominent logical operators.

```
jshell> true || true
$1 ==> true
jshell> true || false
$2 ==> true
jshell> false || true
$3 ==> true
jshell> false || false
$4 ==> false
jshell> true ^ true
$5 ==> false
jshell> true ^ false
$6 ==> true
jshell> false ^ true
$7 ==> true
jshell> false ^ false
$8 ==> false
jshell> !true
$9 ==> false
jshell> !false
$10 ==> true
jshell> int x = 6;
x ==> 6
jshell> !(x > 7)
$11 ==> true
jshell>
```

Summary

In this step, we:

- We're introduced to the `boolean` primitive type
- Understood where logical operators are used in Java programs
- Explored the truth-tables of commonly used logical operators

Step 09: Short-Circuit Evaluation (With Puzzles)

Consider code below. The expression `j > 15 && i++ > 5` evaluates to `false` as expected. `j>15` is `false` as `j` has a value of `15`.

```
jshell> int j = 15;
j ==> 15
jshell> int i = 10;
i ==> 10
jshell> j > 15 && i++ > 5
$1 ==> false
jshell> j
j ==> 15
jshell> i
i ==> 10
```

You can also observe that the value of i remains unchanged `10`.

Why? Because `i++ > 5` was not even evaluated. Why? `&&` is lazy. It saw that `j > 15` is false. Irrespective of the result of second expression, the result of this `&&` would be false. So, it does NOT evaluate the second expression.

A more detailed explanation

The expression `j > 15 && i++ > 5` was scanned from left to right. As the first operand to `&&` evaluated to `false`, the compiler got lazy. `&&` avoids evaluating expressions that don't affect the final result. The optimization has a name: **Short-Circuit Evaluation**, also called **lazy evaluation**.

The logical operator `&` is another version of the logical **AND** operation, that does away with lazy evaluation.

Both operands to `&` are always evaluated.

```
jshell> j > 15 & i++ > 5
$1 ==> false
jshell> j
j ==> 15
jshell> i
i ==> 11
jshell>
```

Similarly, the logical **OR** operator also has two versions:

- The `||` operator we saw earlier. This exhibits lazy evaluation.
- The `|` operator, without lazy evaluation.

It is bad programming practice for our code to depend on the compiler's lazy evaluation. It makes code less readable, and can hide difficult-to-fix software bugs. It obviously adds to the code maintenance burden, so don't do it unless you like being in your peers' bad books.

Summary

In this step, we:

- Examined conditions involving logical operators, that had lazy evaluation
- Observed that the lazy evaluation depends on the operator's truth-table
- Saw versions of the operators without lazy evaluation
- Learned that code depending on lazy-evaluation, is less readable

Step 10: Character Types

Earlier, we explored how we could store basic keyboard characters(called **ascii-code** characters), such as:

- Upper-Case and lower-case letters (A-Z, a-z)
- Numeric characters (0-9)
- Punctuation and other special characters (such as ';', '$', '{', etc.)

Turns out Java supports a much larger family of character encoding sets, called **Unicode**. All Unicode characters can be input to, understood and processed by, as well as output from your code.

Snippet-01 : Unicode characters

Not all Unicode characters can be input from your keyboard. But Java allows you to handle their values from your code, if you deal correctly with their format.

```
jshell> char ch = '"';
ch ==> '"'
jshell> char c = '\u0022';
c ==> '"'
```

Integer values can be stored in `char` variables. If the value is within a meaningful range, it also corresponds to the **ascii** value of a keyboard character.

```
jshell> char cn = 65;
cn ==> 'A'
```

Integer arithmetic can be performed on `char` data.

```
jshell> cn++
$1 ==> 'A'
jshell> cn
$2 ==> 'B'
jshell> ++cn
$3 ==> 'C'
jshell> cn + 5
$4 ==> 72
jshell> cn
cn ==> 'C'
jshell> (int)cn
cn ==> 67
jshell>
```

Summary

In this step, we:

- Were introduced the the `char` data type
- Learned that Unicode takes the character set beyond your keyboard
- An ascii character is `char` value, encoded by an integer value

Step 11: Programming Exercise PE-02

Exercise Set

1. Write a Java class `MyChar` that is a special type of `char`. An object of type `MyChar` is created round an input `char` data element, and has operations that do the following:
    - Check if the input character is a:
        - Numeric Digit
        - Letter of the Alphabet
        - Vowel (Either upper-case or lower-case)
        - Consonant (Either upper-case or lower-case) NOTE: A letter is a consonant if not a vowel
    - Print all the letters of the Alphabet in
        - Upper-Case
        - Lower-Case

In Essence, a runner `class` for `MyChar` would have its `main` method run code similar to:

```
MyChar myChar = new MyChar('c');
System.out.println(myChar.isDigit());
System.out.println(myChar.isAlphabet());
System.out.println(myChar.isVowel());
System.out.println(myChar.isConsonant());
myChar.printLowerCaseAlphabets();
myChar.printUpperCaseAlphabets();
```

Step 12: Solution To PE-02, Part 1 - `isVowel()`

MyCharRunner.java

```
package com.in28minutes.primitive.datatypes;

public class MyCharRunner {
    public static void main(String[] args) {
        MyChar myChar = new MyChar('c');
        System.out.println(myChar.isVowel());
    }
}
```

MyChar.java

```
package com.in28minutes.primitive.datatypes;

public class MyChar {
    private char ch;

    public MyChar(char ch) {
        this.ch = ch;
    }

    public boolean isVowel() {
        if(ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
            return true;
        }
        if(ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U') {
            return true;
        }
        return false;
    }
}
```

Console Output :

false

Step 13: Solution To PE-02, Part 2 - `isDigit()`

MyCharRunner.java

```
package com.in28minutes.primitive.datatypes;

public class MyCharRunner {
    public static void main(String[] args) {
        MyChar myChar = new MyChar('c');
        System.out.println(myChar.isDigit());
        System.out.println(myChar.isVowel());
    }
}
```

MyChar.java

```
package com.in28minutes.primitive.datatypes;

public class MyChar {
    private char ch;

    public MyChar(char ch) {
        this.ch = ch;
    }

    public boolean isVowel() {
        if(ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
            return true;
            }

            if(ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U') {
                return true;
            }
            return false;
        }

        public boolean isDigit() {
            if(ch >= 48 && ch <= 57) {
                return true;
            }
            return false;
        }

}
```

Console Output :

false

false

Step 14: Solution To PE-02, Part 3 - Other Methods

MyCharRunner.java

```
package com.in28minutes.primitive.datatypes;

public class MyCharRunner {
    public static void main(String[] args) {
        MyChar myChar = new MyChar('c');
        System.out.println(myChar.isDigit());
        System.out.println(myChar.isAlphabet());
        System.out.println(myChar.isVowel());
        System.out.println(myChar.isConsonant());
        myChar.printLowerCaseAlphabets();
        myChar.printUpperCaseAlphabets();
    }
}
```

MyChar.java

```
package com.in28minutes.primitive.datatypes;

public class MyChar {
    private char ch;

    public MyChar(char ch) {
        this.ch = ch;
    }

    public boolean isVowel() {
        if(ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') {
            return true;
        }
        if(ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U') {
            return true;
        }
        return false;
    }

    public boolean isConsonant() {
        if(isAlphabet() && !(isVowel())) {
            return true;
        }
        return false;
    }

    public boolean isDigit() {
        if(ch >= 48 && ch <= 57) {
            return true;
        }
        return false;
    }

    public boolean isAlphabet() {
        if(ch >= 97 && ch <= 122) {
            return true;
        }
        if(ch >= 65 && ch <= 90) {
            return true;
        }
        return false;
    }

    public void printLowerCaseAlphabets() {
        for(char ch='a'; ch <= 'z'; ch++) {
            System.out.println(ch);
        }
    }

    public void printUpperCaseAlphabets() {
        for(char ch='A'; ch <= 'Z'; ch++) {
            System.out.println(ch);
        }
    }
}
```

Console Output :

```
false

true

false

true

a

b

c

d

e

f

g

h

i

j
```

k

l

m

n

o

p

q

r

s

t

u

v

w

x

y

z

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Step 15: The Primitive Types - A Review

In this section, we built on our knowledge of the primitive Java types.

- We first got familiar with built-in wrappers for the integer types, that store useful type information.
- Going from the integer to the floating-point types, we examined type compatibility, and how the compiler warns you about common pitfalls. We used explicit casts to force type conversions, and learned that implicit type conversions are quite common.
- We moved on to the `BigDecimal` class, a floating-point type with greater precision and accuracy.
- Next in line was `boolean` , where we built on what we know of logical expressions. We focused on the logical operators, more so on short-circuit evaluation of their expressions.
- We saw how dependency on side-effects, and on lazy evaluation, is not a good programming practice.
- Finally, we got to the `char` type, and were pleasantly surprised to know, that the keyboard is not the limit.

Introducing Conditionals - if, switch and more

Decision making is a part of our daily lives. Computers do tasks for humans, so even programs need to make decisions. They do this by checking logical conditions, which evaluate to `boolean` values.

In the following steps, we will explore the following **conditional** statements:

- `if`
- `if - else`
- `if - else if - else`
- `switch`

What better way to master these basics, than using them to solve a programming challenge?

Here we go!

Programming Challenge : Design A Menu (The *Menu-Challenge*)

- Ask the User for input:
  - Enter two numbers
  - Choose the arithmetic to do on those:
    - Add
    - Subtract
    - Multiply
    - Divide

- Perform the Operation
- Publish the Result

An example scenario could be:

Enter First Number

2

Enter Second Number

4

1 - Add

2 - Subtract

3 - Multiply

4 - Divide

Enter your choice of operation

3

The Result Is : 8

Summary

In this step, we:

- Discussed how input affects the control-flow of a program
- Listed out the conditionals available with Java
- Selected a programming challenge to help us learn about conditions

Step 01: The `if` and `if - else` Conditionals

An `if` statement is the most basic way to manage control-flow in a program. A `boolean` condition is tested, and if found to be `true` , some code is executed. Otherwise, that code is not run.

- "*Do Something when `condition` is `true`* "

Conceptually, the `if` statement looks like this:

```
    if(condition) {
        <if-body>
    }
```

An `if - else` statement improves over the plain `if` . We can now choose between executing two pieces of code, depending on what `condition` evaluates to.

- "*Do Something when `condition` is `true` , something else when `condition` is `false`* "

An `if - else` statement boils down to the following:

```
    if(condition) {
        <if-body>
    } else {
        <else-body>
    }
```

If `condition` is found to be `true`, the statement block `<if-body>` is executed, otherwise `<else-body>` is run.

Let's now look at a examples that make use of these conditionals.

Snippet-01 : `if` behavior

In this example:

- we have used compound comparison operators such as `<=` and `>=`, which also evaluate to `boolean` values.
- Also used, are logical operators such as `&&` and `||`, which both accept and return values of type `boolean`.

```
jshell> int i = 3;
i ==> 3
jshell> if(i==3) {
   ..>> System.out.println("True");
   ..>> }
True
jshell> if(i<2) {
   ..>> System.out.println("True");
   ..>> }

jshell> if(i<=3 || i >= 35) {
   ..>> System.out.println("True");
   ..>> }
True
jshell> if(i<=3 && i >= 35) {
   ..>> System.out.println("True");
   ..>> }
```

`if - else` allows us to specify code to execute when a condition is `false`.

```
jshell> if(i==3) {
   ..>> System.out.println("True");
   ..>> } else {
   ..>> System.out.println("i is not 3");
   ..>> }
True
jshell> i = 5;
i ==> 5
jshell> if(i==3) {
   ..>> System.out.println("True");
   ..>> } else {
   ..>> System.out.println("i is not 3");
   ..>> }
i is not 3
jshell>
```

Snippet-02 : chained `if - else` - v1

IfStatementRunner.java

```
com.in28minutes.ifstatement.examples;

public class IfStatementRunner {
    public static void main(String[] args) {
        //int i=25;
        int i=26;
        if(i == 25) {
            System.out.println("i = 25");
        } else {
            System.out.println("i != 25");
        }
    }
}
```

Console Output

i != 25

Snippet-03 : chained if-else v2

We would want to test a number for 3 conditions - `value is 24`, `value is 25` or `value is not 24 and 25`. Let's consider the code from the example below.

IfStatementRunner.java

```

    com.in28minutes.ifstatement.examples;

    public class IfStatementRunner {
        public static void main(String[] args) {
            // if i is 25, print i = 25
            //if i is 24, print i = 24
            //otherwise, print i != 25 and i != 24

            int i=25;
            if(i == 25) {
                System.out.println("i = 25");
            }
            if(i == 24) {
                System.out.println("i = 24");
            } else {
                System.out.println("i != 25 and i != 24");
            }
        }
    }
```

Console Output

i = 25

i != 25 and i != 24

Snippet-03 Explained

What just happened here? The value of `i` is set to `25` within the program, so like in the previous example, only ***i = 25*** should have been printed.

Why the extra print?

We started off trying to check fro 3 possibilities: * `i` being equal to `25` * `i` being equal to `24` * Neither of the above

We get wrong output, because `i == 25` in first `if` is independent from `i == 24` in the `if - else` that follows it.

Our decision making is not continuous. We need a tighter conditional to deal with more than two alternatives. We will explore this topic in the next step.

Summary

In this step, we:

- Learned about the `if` and `if - else` conditionals
- Tried out a few examples to see how they are used

Step 02: The `if - else if - else` Conditional

The `if - else if - else` statement solves the issue we faced, while trying to test more than two conditions.

Let's see an example.

Snippet-01 : Matching The `if` Clause

IfStatementRunner.java

```
com.in28minutes.ifstatement.examples;

public class IfStatementRunner {
    public static void main(String[] args) {
        // if i is 25, print i = 25
        //if i is 24, print i = 24
        //otherwise, print i != 25 and i != 24

        int i=25;
        if(i == 25) {
            System.out.println("i = 25");
        } else if(i == 24) {
            System.out.println("i = 24");
        } else {
            System.out.println("i != 25 and i != 24");
        }
    }
}
```

Console Output

i = 25

Let's now try giving `i` a different value, say `24` .

Snippet-02: Matching The `else if` Clause

IfStatementRunner.java

```
com.in28minutes.ifstatement.examples;

public class IfStatementRunner {
    public static void main(String[] args) {
        // if i is 25, print i = 25
        //if i is 24, print i = 24
        //otherwise, print i != 25 and i != 24

        //int i=25;
        int i=24;
        if(i == 25) {
            System.out.println("i = 25");
        } else if(i == 24) {
            System.out.println("i = 24");
        } else {
            System.out.println("i != 25 and i != 24");
        }
    }
}
```

Console Output

i = 24

Snippet-02 Explained

- With `i` holding `24`, a different condition (the one coresponding to `i == 24`) evaluates to `true`.

Let's now try to get a match with the `else` clause, the only one unexplored so far.

Snippet-03: Matching The `else` Clause

IfStatementRunner.java

```java
com.in28minutes.ifstatement.examples;

public class IfStatementRunner {
    public static void main(String[] args) {
        // if i is 25, print i = 25
        //if i is 24, print i = 24
        //otherwise, print i != 25 and i != 24

        //int i=24;
        int i=26;
        if(i == 25) {
            System.out.println("i = 25");
        } else if(i == 24) {
            System.out.println("i = 24");
        } else {
            System.out.println("i != 25 and i != 24");
        }
    }
}
```

Console Output

i != 25 and i != 24

Snippet-03 Explained

When `i` is given a value of `26`, the first two conditions are false. Hence, the code in the `else` gets executed.

***one, and only one** clause among those present in an `if` - `else if` - `else` statement ever evaluates to `true`. Also, the code block corresponding to the matched clause will get executed. This is ensured even if conditions are duplicated, or overlap. In that scenario, only the first such condition, downward from the `if` in sequence, will evaluate to `true`. The remaining possible matches are not even checked.*

Summary

In this step, we:

- Learned about the `if` - `else if` - `else` conditional
- Found out how to test when each different clause gets executed
- Understood the guarantees made by Java regarding the conditional's code execution

Step 03: Puzzles on `if`

Try and guess the outputs of the following puzzles.

Programming Puzzle PP-01

```
public class puzzleRunner {

    public static void main(String[] args) {
        puzzleOne();
    }

    public static void puzzleOne() {
        int k = 15;
        if(k > 20) {
            System.out.println(1);
        } else if(k > 10) {
            System.out.println(2);
        } else if(k < 20) {
            System.out.println(3);
        } else {
            System.out.println(4);
        }
    }
}
```

Answer:

2

Programming Puzzle PP-02

```
public class puzzleRunner {
    public static void main(String[] args) {
        puzzleTwo();
    }

    public static void puzzleTwo() {
        int l = 15;
        if(l < 20)
        System.out.println("l < 20");
        if(l > 20)
        System.out.println("l > 20");
        else
        System.out.println("Who Am I?");
    }
}
```

Answer:

l < 20

Who Am I?

Programming Puzzle PP-03

```
public class puzzleRunner {
    public static void main(String[] args) {
        puzzleThree();
    }

    public static void puzzleThree() {
        int m = 15;
        if(m > 20)
        if(m < 20)
        System.out.println("m >  20");    else
        System.out.println("Who Am I?");
        }
    }
```

Answer:

> Nothing is printed. Because the code is structured this way.

```
if(m > 20) {
    if(m < 20)
        System.out.println("m >  20");
    else
        System.out.println("Who Am I?");
}
```

Programming Puzzle PP-04

```
jshell> int i = 0;
i ==> 3
jshell> if(i) {
   ..>> System.out.println("i");
   ..>> }
| Error:
| incompatible types: int cannot be converted to boolean
| if(i)
|____^
jshell> if(i=1) {
   ..>> System.out.println("i");
   ..>> }
| Error:
| incompatible types: int cannot be converted to boolean
| if(i=1)
|____^-^
jshell> if(i==1) {
   ..>> System.out.println("i");
   ..>> }
jshell>
```

Answer:

Compiler Error

Programming Puzzle PP-05

```
public class puzzleRunner {
    public static void main(String[] args) {
        puzzleFive();
    }

    public static void puzzleFive() {
        int number = 5;
        if(number < 0)
        number = number + 10;
        number++;
        System.out.println(number);
    }
}
```

**Answer : **

6

> In the absence of explicit blocks, Only the statement next to the if statement is considered to be part of the if statement block.

Step 04: Reading User Input

Look at the statement of *Menu-Challenge* once again:

Menu-Challenge

- Ask the User for input:
  - Enter two numbers
  - Choose an Arithmetic Operation to perform on them:
    - Add
    - Subtract
    - Multiply
    - Divide
- Perform the Operation
- Publish the Result

We have a good idea about how the `if - else if - else` conditional works.What we don't know, however, is how a such a conditional would behave when fed with random input. To test those scenarios out, the next step would be to learn how to take console input, from within our code.

We will do just that, in our next example.

Snippet-01: Reading console input

Java provides a built-in `class` named `Scanner`, to scan user input from the console. Roping in this utility would require you, the programmer, to do the following:

- `import` the `java.util.Scanner` `class` within your code (here, we were writing code in the Eclipse IDE)
- Create a `scanner` object which is of type `Scanner`. This involves invoking the `Scanner` constructor through the `new` operator. You also need to pass `System.in` as a constructor parameter, which ties `scanner` to the console input.
- To read integer input from the console, call the method `scanner.nextInt()`. The keyboard's key needs to be pressed to complete user input. That number is passed on to your code, where it can be passed around.

MenuScanner.java

```
package com.in28minutes.ifstatement.examples;
import java.util.Scanner;

public class MenuRunner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter Number1: ");
        int number1 = Scanner.nextInt();
        System.out.println("The number you entered is: " + number1);
    }
}
```

Console Output

Enter Number1: 35

The number you entered is: 35

Summary

In this step, we:

- Revisited our *Menu-Challenge* problem statement, and found we needed to read user input
- Explored the basic usage of the `Scanner` utility to fulfill this need

Step 05: *Menu-Challenge* : Reading More Input

The *Menu-Challenge* does not stop at a single user input. It requires a total of three numbers to be typed in at the console. So how do we continue asking for input, and read them when they are given?

Snippet-01 : Implementing *Menu-Challenge*

MenuScanner.java

```java
package com.in28minutes.ifstatement.examples;
import java.util.Scanner;

public class MenuRunner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter Number1: ");
        int number1 = Scanner.nextInt();
        System.out.print("Enter Number2: ");
        int number2 = Scanner.nextInt();

        System.out.println("Select The Operation Choice");
        System.out.println("1 - Add");
        System.out.println("2 - Subtract");
        System.out.println("3 - Multiply");
        System.out.println("4 - Divide");
        System.out.print("Enter Choice: ");
        int choice = Scanner.nextInt();

        System.out.print("Your Inputs Are:");
        System.out.println("Number1: " + number1);
        System.out.println("Number2: " + number2);
        System.out.println("Choice: " + choice);
    }
}
```

Console Output

Enter Number1: 25

Enter Number2: 50

Operation Choices Available

1 - Add

2 - Subtract

3 - Multiply

4 - Divide

Enter Choice: 4

Your Inputs Are

Number1: 25

Number2: 50

Choice: 4

Snippet-01 Explained

Repeatedly calling `scanner.nextInt()` will keep reading in any number the user may input. Also, the user needs to press the keyboard key to send each input.

We were not only able to read in all three values we need, but also wrote them out on the console. The user can now see that we got his data!

Summary

In this step, we:

- Figured out how to read more than one input from the console
- Demonstrated how values read in, can be preserved and used within the program

Step 06: *Menu-Challenge* - Reading Input, Computing Result, Displaying Output

We don't think that after the previous step, anything can stop you from completing the *Menu-Challenge*. Here is one such way, from head-to-toe.

Snippet-01 : All computations

Our solution above does a very simple thing. It combines the mechanisms we learned for reading input and testing conditions, to solve *Menu-Challenge*.

The `if` - `else` - `else if` statement has a total of `5` clauses:

- To check for `4` favorable conditions (The `choice` values for the `4` supported operations). One `if` and three `else` - `if` clauses do the stuff for us.
- the last default condition, which corresponds to a `choice` value not supported, is handled by an `else` clause.

MenuScanner.java

```java
package com.in28minutes.ifstatement.examples;
import java.util.Scanner;

public class MenuRunner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter Number1: ");
        int number1 = Scanner.nextInt();
        System.out.print("Enter Number2: ");
        int number2 = Scanner.nextInt();

        System.out.println("Select The Operation Choice");
        System.out.println("1 - Add");
        System.out.println("2 - Subtract");
        System.out.println("3 - Multiply");
        System.out.println("4 - Divide");
        System.out.print("Enter Choice: ");
        int choice = Scanner.nextInt();

        System.out.print("Your Inputs Are:");
        System.out.println("Number1: " + number1);
        System.out.println("Number2: " + number2);
        System.out.println("Choice: " + choice);

        if(choice == 1) {
            System.out.println("Result = " + (number1 + number2));
        } else if(choice == 2) {
            System.out.println("Result = " + (number1 - number2));
        } else if(choice == 3) {
            System.out.println("Result = " + (number1 * number2));
        } else if(choice == 4) {
            System.out.println("Result = " + (number1 / number2));
        } else {
            System.out.println("Invalid Operation");
        }
    }
}
```

Console Output

Enter Number1: 23

Enter Number2: 10

Operation Choices Available

1 - Add

2 - Subtract

3 - Multiply

4 - Divide

Enter Choice: 2

Your Inputs Are

Number1: 23

Number2: 10

Choice: 2

Result = 13

Console Output

Enter Number1: 25

Enter Number2: 35

Operation Choices Available

1 - Add

2 - Subtract

3 - Multiply

4 - Divide

Enter Choice: 5

Your Inputs Are

Number1: 23

Number2: 10

Choice: 5

Invalid Operation

Summary

In this step, we:

- Combined our knowledge of conditionals, with our recent learning on taking multiple console inputs
- Ultimately solved the *Menu-Challenge* problem

Step 07: Introducing `switch`

If you remember, our initial list of conditionals to manage control-flow, also had a `switch` statement. A `switch` also tests multiple conditions, and just like an `else` clause, it can handle the default possibility. Conceptually, a `switch` statement looks like the following:

```
switch(condition) {
    case 1:
        //<statements>
        break;
    case 2:
        //<statements>
        break;
    //...
    default:
        //<statements>
        break;
    }
}
```

`default` clause is executed when none of the cases match.

`break;` statement is used to break out of the switch after a successful match.

Let's look at a few examples on how to use `switch` .

Snippet-01: The `switch` statement

Using a `switch` leads to code that is quite readable, doesn't it? Compare it with the chained `if` - `else if` - `else` statements we used in the previous step.

Leaving out the `break` statement from every `case` clause is a very common error. It leads to a situation called **switch-fall-through**. Here, even if there is a match with a particular `case` clause, all clauses following this one are executed in sequence, until a `break` is encountered somewhere!

```
jshell> int i = 5;
i ==> 5
jshell> switch(i) {
   ..>> case 1 : System.out.println("1");
   ..>> case 5 : System.out.println("5");
   ..>> default : System.out.println("default");
   ..>> }
5
default
jshell> i = 1;
i ==> 1
jshell> switch(i) {
   ..>> case 1 : System.out.println("1");
   ..>> case 5 : System.out.println("5");
   ..>> default : System.out.println("default");
   ..>> }
1
5
default
```

Adding `break` s ensures that only the matching case is executed.

```
jshell> switch(i) {
   ..>> case 1 : System.out.println("1"); break;
   ..>> case 5 : System.out.println("5"); break;
   ..>> default : System.out.println("default"); break;
   ..>> }
1
jshell>
```

Snippet-02: refactoring MenuScanner

Let's now refactor the `MenuScanner` example to use `switch` statement.

MenuScanner.java

```java
package com.in28minutes.ifstatement.examples;
import java.util.Scanner;

public class MenuRunner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter Number1: ");
        int number1 = Scanner.nextInt();
        System.out.print("Enter Number2: ");
        int number2 = Scanner.nextInt();

        System.out.println("Select The Operation Choice");
        System.out.println("1 - Add");
        System.out.println("2 - Subtract");
        System.out.println("3 - Multiply");
        System.out.println("4 - Divide");
        System.out.print("Enter Choice: ");
        int choice = Scanner.nextInt();

        System.out.print("Your Inputs Are:");
        System.out.println("Number1: " + number1);
        System.out.println("Number2: " + number2);
        System.out.println("Choice: " + choice);

        //performOperationUsingChainedIfElse(number1, number2, choice);`
        performOperationUsingSwitch(number1, number2, choice);
    }

    public static void performOperationUsingSwitch(int number1, int number2, int choice) {
        switch(choice) {
            case 1 : System.out.println("Result = " + (number1 + number2)); break;
            case 2 : System.out.println("Result = " + (number1 - number2)); break;
            case 3 : System.out.println("Result = " + (number1 * number2)); break;
            case 4 : System.out.println("Result = " + (number1 / number2)); break;
            default : System.out.println("Invalid Operation"); break;
        }
    }

    public static void performOperationUsingChainedIfElse(int number1, int number2, int choice) {
        if(choice == 1) {
            System.out.println("Result = " + (number1 + number2));
        } else if(choice == 2) {
            System.out.println("Result = " + (number1 - number2));
        } else if(choice == 3) {
            System.out.println("Result = " + (number1 * number2));
        } else if(choice == 4) {
            System.out.println("Result = " + (number1 / number2));
        } else {
            System.out.println("Invalid Operation");
        }
    }
}
```

Console Output

Enter Number1: 23

Enter Number2: 10

Operation Choices Available

1 - Add

2 - Subtract

3 - Multiply

4 - Divide

Enter Choice: 2

Your Inputs Are

Number1: 23

Number2: 10

Choice: 2

Result = 13

Console Output

Enter Number1: 25

Enter Number2: 35

Operation Choices Available

1 - Add

2 - Subtract

3 - Multiply

4 - Divide

Enter Choice: 5

Your Inputs Are

Number1: 23

Number2: 10

Choice: 5

Invalid Operation

Summary

In this step, we:

- Explored the `switch` conditional
- Saw how it could implement the same control-flow as the `if` family of conditionals
- Learned the importance of coding it correctly, to enjoy Java control-flow guarantees

Puzzles On `switch`

Let's now have some fun with the various conditionals. These puzzles will not only ensure you're still wide awake, they will also give you ample food for thought.

Programming Puzzle PP-01

```
public class SwitchPuzzleRunner {
    public static void main(String[] args) {
        puzzleOne();
    }

    public static void puzzleOne() {
        int number = 2;
        switch(number) {
            case 1:
                System.out.println(1);
            case 2:
                System.out.println(2);
            case 3:
                System.out.println(3);
            default:
                System.out.println("default");
        }
    }
}
```

Answer:

2

3

default

Programming Puzzle PP-02

```
public class SwitchPuzzleRunner {
    public static void main(String[] args) {
        puzzleTwo();
    }

    public static void puzzleTwo() {
        int number = 2;
        switch(number) {
            case 1:
                System.out.println(1);
            case 2:
            case 3:
                System.out.println("Number is 2 or 3");
                break;
            default:
                System.out.println("default");
                break;
        }
    }
}
```

Answer:

Number is 2 or 3

Programming Puzzle PP-03

```
    public class SwitchPuzzleRunner {
        public static void main(String[] args) {
            puzzleThree();
        }

        public static void puzzleThree() {
            int number = 10;
            switch(number) {
                case 1:
                    System.out.println(1);
                    break;
                case 2:
                    System.out.println(2);
                    break;
                case 3:
                    System.out.println(3);
                    break;
                default:
                    System.out.println("default");
                    break;
            }
        }
    }
```

Answer:

default

Programming Puzzle PP-04

```
    public class SwitchPuzzleRunner {
        public static void main(String[] args) {
        puzzleFour();
    }

        public static void puzzleFour() {
            int number = 10;
            switch(number) {
                System.out.println("default");
                break;
                case 1:
                    System.out.println(1);
                    break;
                case 2:
                    System.out.println(2);
                    break;
                case 3:
                    System.out.println(3);
                    break;
                default:
            }
        }
    }
```

Answer:

default

Programming Puzzle PP-05

```
    public class SwitchPuzzleRunner {
        public static void main(String[] args) {
            puzzleFive();
        }

        public static void puzzleFive() {
            long l = 15;
            switch(l) {
            }
        }
    }
```

Answer:

Compiler Error

Programming Puzzle PP-06

```
    public class SwitchPuzzleRunner {
        public static void main(String[] args) {
            puzzleSix();
        }

        public static void puzzleSix() {
            int number = 10;
            int i = number * 2;
            switch(number) {
                case number>5 : System.out.println("number>5");
            }
        }
    }
```

Answer:

Compiler Error

Step 09: Comparing The `if` Family, And `switch`

Let's compare and contrast these two approaches, to gain some experience on how to choose a conditional.

First comes an example using the `if` - `else if` - `else` statement.

Snippet-01 : Formatted Output Using `if`

OperatorChoiceRunner.java

```
    package com.in28minutes.ifstatement.examples;

    public class OperatorChoiceRunner {
```

```
    public static void main(String[] args) {
        OperatorChoice opChoice = new OperatorChoice(5, 2, 1);
        opChoice.operate();
    }
}
```

OperatorChoice.java

```
package com.in28minutes.ifstatement.examples;

public class OperatorChoice {
    private int number1;
    private int number2;
    private int choice;

    public OperatorChoice(int number1, int number2, int choice) {
        this.number1 = number1;
        this.number2= number2;
        this.choice = choice;
    }

    public void operate() {
        System.out.printf("number1 : %d", number1).println();
        System.out.printf("number2 : %d", number2).println();
        System.out.printf("choice : %d", choice).println();

        if(choice == 1) {
            System.out.printf("result : %d", number1 + number2).println();
        } else if(choice == 2) {
            System.out.printf("result : %d", number1 - number2).println();
        } else if(choice == 3) {
            System.out.printf("result : %d", number1 * number2).println();
        } else if(choice == 4) {
            System.out.printf("result : %d", number1 / number2).println();
        } else {
            System.out.println("Invalid Operation");
        }
    }
}
```

Console Output

number1 : 5

number2 : 2

choice : 1

result : 7

Next in line, is an example involving a `switch` .

Snippet-02 : Formatted Output Using `switch`

OperatorChoiceRunner.java

```
package com.in28minutes.ifstatement.examples;

public class OperatorChoiceRunner {
    public static void main(String[] args) {
        OperatorChoice opChoice = new OperatorChoice(5, 2, 1);
        opChoice.operateUsingSwitch();
        }
    }
```

OperatorChoice.java

```
    package com.in28minutes.ifstatement.examples;

    public class OperatorChoice {
        private int number1;
        private int number2;
        private int choice;

        public OperatorChoice(int number1, int number2, int choice) {
            this.number1 = number1;
            this.number2= number2;
            this.choice = choice;
        }

        public void operateUsingSwitch() {
            System.out.printf("number1 : %d", number1).println();
            System.out.printf("number2 : %d", number2).println();
            System.out.printf("choice : %d", choice).println();

            switch(choice) {
                case 1: System.out.printf("result : %d", number1 + number2).println();
                    break;
                case 2: System.out.printf("result : %d", number1 - number2).println();
                    break;
                case 3: System.out.printf("result : %d", number1 * number2).println();
                    break;
                case 4: System.out.printf("result : %d", number1 / number2).println();
                    break;
                default: System.out.printf("result : %d", number1 + number2).println();
                    break;
            }
        }
    }
```

Console Output

number1 : 5

number2 : 2

choice : 1

result : 7

Summary

In this step, we:

- Observed that the same conditional code could be written using an `if` -family conditional, or the `switch` .
- Learned that an `if` family conditional is difficult to get wrong, as the rules for it are very strict. It can be used to evaluate only `boolean` conditions. But it is verbose, and often less readable.
- Came to know that a `switch` conditional can be used to check for only integer values. It is very compact, and very readable. However, the relative order of `case` clauses and `default` are not fixed, and the usage of `break` is optional. This can lead to subtle errors in your program.

Step 10: Programming Exercise PE-03

1. Write a Java `class` which has the following operations within it:

- Given a number between 0 (Sunday) and 6 (Saturday), return if the day is a Weekday
- Given a number between 1 (January) and 12 (December), return the corresponding English name for that month
- Given a number between 1 (January) and 12 (December), return the corresponding English name for that day of the week

Solution to PE-03

CalendarSwitchRunner.java

```
package com.in28minutes.ifstatement.examples;

public class CalendarSwitchRunner {
    public static void main(String[] args) {
        System.out.println(determineNameOfDay(1));
        System.out.println(isWeekDay(1));
        System.out.println(determineMonthOfYear(1));
    }

    public String determineNameOfDay(int dayNumber) {
        switch(dayNumber) {
            case 0 : return "Sunday";
                break;
            case 1 : return "Monday";
                break;
            case 2 : return "Tuesday";
                break;
            case 3 : return "Wednesday";
                break;
            case 4 : return "Thursday";
                break;
            case 5 : return "Friday";
                break;
            case 6 : return "Saturday";
                break;
            default : return "Invalid Day";
                break;
        }
    }

    public String determinenameofMonth(int monthNumber) {
        switch(monthNumber) {
            case 1 : return "January";
            case 2 : return "February";
            case 3 : return "March";
            case 4 : return "April";
            case 5 : return "May";
            case 6 : return "June";
            case 7 : return "July";
            case 8 : return "August";
            case 9 : return "September";
            case 10 : return "October";
            case 11 : return "November";
            case 12 : return "December";
            default : return "Invalid Month";
        }
    }

    public boolean isWeekDay(int dayNumber) {
        switch(dayNumber) {
            case 1 :
            case 2 :
            case 3 :
            case 4 :
            case 5 : return true;
```

```
                case 0 :
                case 6 :
                default : return false;
            }
        }
    }
```

Step 12: Introducing `?:` , The Ternary Operator

The ternary operator `?:` is a logical operator, that works on three operands. Its functioning is similar to an `if` - `else` statement (Checking for just `2` conditions). Exactly one of the two expressions is guaranteed to match.

Conceptually, its syntax is this:

`result = (condition ? expression-if-condition-true : expression:if-condition-false);`

Snippet-01 : ternary operator

`?:` is easy to use. A few examples below:

```
    jshell> boolean isEven;
    isEven ==> false
    jshell> int i = 6;
    jshell> isEven = ( i%2==0 ? true : false);
    isEven ==> true
    jshell> i = 7;
    i ==> 7
    jshell> isEven = ( i%2==0 ? true : false);
    isEven ==> false
```

You can return non boolean values.

```
    jshell> String ifEven = ( i%2==0 ? "YES" : "NO");
    ifEven ==> "NO"
    jshell> i = 6;
    i ==> 6
    jshell> ifEven = ( i%2==0 ? "YES" : "NO");
    ifEven ==> "YES"
```

Both the expressions should return value of same type.

```
    jshell> ifEven = ( i%2==0 ? "YES" : 4 );
    | Error:
    | incompatible types: bad type in conditional expression
    | int cannot be converted to java.lang.String
    | ifEven = ( i%2==0 ? "YES" : 4 );
    |_____________________________^
    jshell> ifEven = ( i%2==0 ? "YES" : true );
    | Error:
    | incompatible types: bad type in conditional expression
    | boolean cannot be converted to java.lang.String
    | ifEven = ( i%2==0 ? "YES" : true );
    |_____________________________^
    jshell>
```

Summary

In this step, we:

- Discovered a more compact Java conditional, the `?:` operator
- Saw that in most cases it behaves like an `if` - `else` construct

Loops

TODO

Revisiting Java loop constructs: `for` and `while`

If you may recall, the structure of a `for` loop is:

```
for(initialization; condition; update) {

     //<Statements Body>

}
```

The `<Statements Body>` inside the loop is executed so long as the `condition` is `true` . Let's look at a few puzzles to explore how we can utilize them.

Snippet 1 : First for loop puzzle

```
jshell> for(int i=0; i <= 10; i++) {

__**...>>** System.out.print(i + " ");

__**...>>** }

0 1 2 3 4 5 6 7 8 9 10

jshell> for(int i=0; i <= 10; i = i+2) {

__**...>>** System.out.print(i + " ");

__**...>>** }

0 2 4 6 8 10

jshell> for(int i=1; i <= 10; i = i+2) {

__**...>>** System.out.print(i + " ");

__**...>>** }

1 3 5 7 9

jshell> for(int i=11; i <= 10; i = i+2) {

__**...>>** System.out.print(i + " ");

__**...>>** }

jshell> for(int i=11; i <= 20;) {

__**...>>** System.out.print(i + " ");

__**...>>** i++;

__**...>>** }

11 12 13 14 15 16 17 18 19 20

jshell> int i = 20;
```

i ==> 20

jshell> `for(i <= 30; i++) {`

__**...>>** `System.out.print(i + " ");`

__**...>>** `}`

21 22 23 24 25 26 27 28 29 30

jshell>

Snippet-1 Explained

All the three control components of a `for` loop are dispensable

- `initialziation`
- `condition`
- `update`

Reference Types

Step 01: Introducing Reference Types

What happens in the background when we create objects?

```
jshell> class Planet {
..>>}
| created class Planet
jshell> Planet jupiter = new Planet();
jupiter ==> Planet@31a5c39e
```

Where is the object `jupiter` stored?

All Java objects are created on the `Heap` or `Heap Memory` .

When we create an new instance of `Planet', it is created on the Heap.

```
jshell> new Planet()
$18 ==> Planet@3f49dace
```

`new Planet()` creates an object on the Heap. In above example `Planet@3f49dace` , the object is stored on the Heap at address `3f49dace` .

```
jshell> Planet jupiter = new Planet();
jupiter ==> Planet@31a5c39e
```

`Planet jupiter = new Planet()` does two things.

- Creates an object on the Heap `new Planet()` . In above example, object is created at Heap location `31a5c39e`
- Stores the reference of the object on the Heap in a variable `jupiter` . In above example, `31a5c39e` is the value stored in variable `jupiter` . That's the memory location where the new object was created.

Let's consider another example.

```
jshell> class Animal {
   ..>> int id;
   ..>> public Animal(int id) {
   ..>> this.id = id;
   ..>> }
   ..>> }
| created class Animal
jshell> Animal dog = new Animal(12);
dog ==> Animal@27c20538
jshell> Animal cat = new Animal(15);
cat ==> Animal@6e06451e
jshell>
```

Two new `Animal` objects are created on the `Heap`. Their memory locations (`references`) are stored in the reference variables - `dog` and `cat` .

In Java, all classes are also called Reference Types. Except for primitive variable instances, all the instances or objects are stored on the Heap. The references to the objects are stored in the reference variables like `jupiter` , `dog` and `cat` .

Summary

In this step, we:

- Looked at what references are
- Had a look at what the contents of a reference variable look like

Step 02: References: Usage And Puzzles

Let's spend some time playing with reference variables.

References that are not initialized by the programmer, are initialized by the Java compiler to `null` . `null` is a special value that stands for an **empty location**. In other words, the `Animal nothing` refers to nothing!

```
jshell> class Animal {
   ..>> int id;
   ..>> public Animal(int id) {
   ..>> this.id = id;
   ..>> }
   ..>> };
| created class Animal
jshell> Animal nothing;
nothing ==> null
```

Assigning the reference `cat` to `nothing` does what one would expect: it assigns the address of the object created with `new Animal(15)` (stored in `cat`), to the variable `nothing` .

```
jshell> Animal dog = new Animal(12);
dog ==> Animal@27c20538
jshell> Animal cat = new Animal(15);
cat ==> Animal@6e06451e_

jshell> nothing = cat;
nothing ==> 6e06451e
```

`nothing` and `cat` are pointing to the same location on the 'Heap'. When we do `nothing.id = 10` we are changing the `id` of the object pointed to by both `nothing` and `cat` .

```
jshell> nothing.id = 10;
$1 => 10
jshell> cat.id
$2 => 10
```

Let's change `nothing` to point to the same object referenced by `dog`.

```
jshell> nothing = dog;
dog ==> 27c20538
jshell> nothing.id;
$3 => 12
jshell>
```

You can `nothing.id` prints the value of the object referenced by `dog` because they are pointing to the same object.

Here are couple of important things to note:

- `nothing = dog` - Assignment between references does not copy the entire referenced object. It only copies the reference. After an assignment, both reference variables point to the same object.
- `nothing.id = 10` - References can be used to modify the objects they reference.

Comparing Reference Variables

Let's look at an example.

With primitive variables, assignment copies values.

```
jshell> int i =5;
i ==> 5
jshell> int j;
j ==> 0
jshell> j = i;
j ==> 5
jshell> j = 6;
j ==> 6
jshell> i;
i ==> 5
```

`j = i` copies the value of `i` into `j`. Later, when value of `j` is changed, `i` is not affected.

Comparing primitive variables compares their values.

```
jshell> i == j;
$4 ==> false
jshell> j = 5;
j ==> 5
jshell> i == j;
$5 ==> true
```

Let's create a few reference variables.

```
jshell> Animal dog = new Animal(12);
dog ==> Animal@4b952a2d
jshell> Animal cat = new Animal(10);
cat ==> Animal@3159c4b8
jshell> Animal ref = cat;
ref ==> Animal@3159c4b8
```

```
jshell> Animal dog2 = new Animal(12);
dog ==> Animal@29ca901e_
```

When we compare reference variables, we are still comparing values. But the values are references - the address of memory locations where objects are stored. The values stored inside the referenced objects are not used for comparison.

Both `cat` and `ref` reference a single `Animal` object created using `new Animal(10)` . `==` returns true

```
jshell> cat == dog;
$6 ==> false
jshell> cat == ref;
$7 ==> true
```

The comparison `dog == dog2` evaluates to `false` since the references i.e. memory locations pointed by these variables are different. They have the same values for `id` field (`12`). But, that is not important!

```
jshell> dog == dog2;
$8 ==> false
jshell>
```

Summary

In this step, we:

- Understood the way reference variables behave during initialization and assignment
- Saw the relevance of a `null` value for references
- Observed how equality of references, is different from equality of values of primitive types

Step 03: Introducing `String`

A sequence of characters, such as `"Hello"` , `"qwerty"` and `"PDF"` is very different from other pieces of data.

In Java, a sequence of characters is typically represented by `String` `class` .

`String` provides several built-in utility methods.

Let's look at a few examples.

```
jshell> "Test".length()
$1 ==> 4
```

`"Test"` is a string literal, so the compiler internally creates an object, gives it the type `String` and allocates memory for it. Since `length()` is a method of `String` , it can be invoked on this `"Test"` object.

In the example below, `str` is a reference to a `String` object, containing the value `"Test"` . Creating a `String` object is an exception to how we typically create objects in Java. You don't need to make a `String` constructor call. Contrast this with how we would create a `BigDecimal` object.

```
jshell> String str = "Test";
str ==> "Test"
jshell> BigDecimal bd = new BigDecimal("1.0");
bd ==> 1.0
```

String indexes, like those of arrays, start at `0` . The `charAt(int)` method takes an index value as its argument, and returns the character symbol present at that index.

```
jshell> str.charAt(0)
$2 ==> 'T'
jshell> str.charAt(2)
$3 ==> 's'
jshell> str.charAt(3)
$4 ==> 't'
```

The `substring()` method returns a `String` reference, and has overloaded versions:

- The `substring(int, int)` method returns the sequence of character symbols starting at the lower index, and ending just before the upper index.
- The `substring(int)` method returns the sequence of character symbols starting from the index to end of string.

```
jshell> String biggerString = "This is a lot of text";
biggerString ==> "This is a lot of text"
jshell> biggerString.substring(5)
$5 ==> "is a lot of text"
jshell> biggerString.substring(5, 13)
$6 ==> "is a lot"
jshell> biggerString.charAt(13)
$7 ==> ' '
jshell>
```

Summary

In this step, we:

- Were introduced to the `String` class, that represents sequences of characters
- Explored a few utility methods of the `String` class

Step 04: Programming Exercise PE-01, And `String` Utilities

Exercise

1. Write a method to print the individual characters of a given text string, separately.

Solution To PE-01

```
jshell> String text = "Equation";
   ..>> for (int i=0; i < text.length(); i++) {
   ..>> System.out.println(text.charAt(i));
   ..>> }
E
q
u
a
t
i
o
n
jshell>
```

Common `String` Utilities

The `String` class is part of the built-in `java.lang` package. It provides several methods for common text processing. Let's have a peek at some of them, shall we?

Snippet-01: `String` Utilities

Here are a few of the methods used in the examples below:

- `indexOf()` : Has two overloaded versions. `indexOf(char)` returns the position where a character occurs in a string, the first time. `indexOf(String)` returns the starting position where a string occurs within our string, the first time.
- `lastIndexOf()` : Similar in function to `indexOf()` , but replace "*first time*" with "**final time**" in its description.
- `startsWith()` : Returns `true` if our string starts with the given *prefix*, `false` otherwise.
- `endsWith()` : Returns `true` if our string ends with the given *suffix*, `false` otherwise.
- `isEmpty()` : Returns `true` if our string is empty, `false` otherwise.
- `equals()` : Returns `true` if our string is identical to the argument, `false` otherwise.
- `equalsIgnoreCase()` : Returns `true` if our string is identical to the argument, ignoring the case of its characters. Will return `false` otherwise.

```
jshell> String someString = "This is a lot of text again";
someString ==> "This is a lot of text again"
jshell> someString.indexOf("lot")
$1 ==> 10
jshell> someString.charAt(10)
$2 ==> 'l'
jshell> someString.indexOf('i')
$3 ==> 2
jshell> someString.lastIndexOf('i')
$4 ==> 25
jshell> someString.contains("text")
$5 ==> true
jshell> someString.startsWith("This")
$6 ==> true
jshell> someString.startsWith("jfsdklfj")
$7 ==> false
jshell> someString.endsWith("in")
$7 ==> true
jshell> someString.endsWith("ain")
$8 ==> true
jshell> someString.endsWith("gain")
$9 ==> true
jshell> someString.endsWith("againasdf")
$10 ==> false
jshell> someString.isEmpty()
$11 ==> false
jshell> "".isEmpty()
$12 ==> true
jshell> "true".equals("true")
$13 ==> true
jshell> String str = "value";
str ==> "value"
jshell> str.equals("value")
$14 ==> true
jshell> str.equals("VALUE")
$15 ==> false
jshell> str.equalsIgnoreCase("VALUE")
$16 ==> true
jshell>
```

Summary

In this step, we:

- Used our `String` programming skills on a small challenge.
- Explored a set of simple, yet useful `String` utilities.

Step 05: `String` Immutability

What picture forms in your mind on hearing the word "**immutable**"? Someone whose voice cannot be muted out? Or is it the other way round?

The word "immutable" is related to the concept of "mutability", or the possibility of "mutating" something.

"**mutate**" means "**to change**". "Immutable" refers to something that "**cannot be changed**".

`String` objects are immutable. You cannot change their value after they are created.

The method `concat()` joins the contents of two `String` objects into one.

```
jshell> String str = "in28Minutes";
str ==> "in28Minutes"
jshell> str.concat(" is awesome")
$1 ==> "in28Minutes is awesome"
```

However, the original value referred by `str` remains unchanged. The `concat` method create a new `String` object.

```
jshell> str
str ==> "in28Minutes"
```

Just like `concat()`, other `String` methods such as `toUpperCase()`, `toLowerCase()` and `trim()` return new `String` objects.

```
jshell> String anotherString = str.concat(" is awesome");
anotherString ==> "in28Minutes is awesome"
jshell> str
str ==> "in28Minutes"
jshell> String string2 = anotherString.concat(".");
string2 ==> "in28Minutes is awesome."
jshell> str
str ==> "in28Minutes"
jshell> anotherString
anotherString ==> "in28Minutes is awesome"
jshell> String s= "in28Minutes is awesome."
s ==> "in28Minutes is awesome."
jshell> s.toUpperCase()
s ==> "IN28MINUTES IS AWESOME."
jshell> s.toLowerCase()"
s ==> "in28minutes is awesome."
jshell> String str2= "  in28Minutes is awesome    "
str2 ==> "  in28Minutes is awesome    "
jshell> str2.trim()
str2 ==> "in28Minutes is awesome"
jshell>
```

Summary

In this step, we:

- We understood that `String` objects are immutable
- Observed how common `String` utilities return a new String.

Step 06: More `String` Utilities

The symbol `+` denotes addition, and addition only, right? Any school kid will tell you that, or laugh at you if you disagree.

Heck, we even saw how to use it with the primitive numeric types, such as `int` , `double` and others related to it.

Java does not strictly follow your arithmetic text book.

`+` can also be used as a `String` concatenation operator.

Here is how `+` works

- if both operands of `+` are numeric, then arithmetic `+` (addition) is performed.
- If any one of the operators is a `String` , then string concatenation is performed.

```
    jshell> 1 + 2
    $1 ==> 3
    jshell> "1" + "2"
    $2 ==> "12"
    jshell> "1" + 2
    $3 ==> "12"
    jshell> "1" + 23
    $4 ==> "123"
    jshell> 1 + 23
    $5 ==> 24
    jshell> "1" + 2 + 3
    $6 ==> "123"
    jshell> 1 + 2 + "3"
    $7 ==> "33"
```

In the example below, a different value is printed when parentheses are used around `i + 20` .

```
    jshell> int i = 20;
    i ==> 20
    jshell> System.out.println("Value is " + i);
    Value is 20
    jshell> System.out.println("Value is " + i + 20);
    Value is 2020
    jshell> System.out.println("Value is " + (i + 20));
    Value is 40
```

`join()` is used to join a set of `String` values.

```
    jshell> String.join(",", "2", "3", "4");
    $8 ==> "2,3,4"
```

`replace(String, String)` replaces all occurrences of the first sub-string with the second one. It also works with `char` data type.

```
    jshell> "abcd".replace('a', 'z');
    $9 ==> "zbcd"
    jshell> "abcd".replace("ab", "xyz");
    $10 ==> "xyzcd"
    jshell>
```

Summary

In this step, we:

- Learned that the `+` operator is overloaded for `String` concatenation
- Observed how `+` interprets its operands, depending on the context
- Noticed a few more `String` utility methods, such as `join()` and `replace()`

Step 07: Storing mutable text

`StringBuffer` and `StringBuilder` allow you to modify a string literal in-place.

```
    jshell> "123" + "123" + "1234" + "12345"
    $1 ==> "123123123412345"
    jshell> StringBuffer sb = new StringBuffer("TEst");
    sb ==> "TEst"
    jshell> sb.append(" 123");
    $2 ==> "TEst 123"
    jshell> sb
    sb ==> "TEst 123"
    jshell> sb.setCharAt(1, 'e');
    sb ==> "Test 123"
    jshell> StringBuilder sbldr = new StringBuffer("TEst");
    sbldr ==> "TEst"
    jshell>
```

How do you choose which one to use?

- `StringBuffer` is thread safe. If you are writing a multi threaded program(more on this in a later section), use `StringBuffer`.
- Otherwise, use `StringBuilder`, since it offers better performance in both execution-time and memory usage.

Summary

In this step, we:

- Learned about `StringBuffer` and `StringBuilder`

Step 08: Introducing Wrapper Classes

Each primitive type in Java has a corresponding built-in **wrapper class**. Wrapper classes are also immutable (As well as `final`, more on this a little later).

Following is a list of built-in Java wrapper classes and their corresponding primitive types:

- `byte : Byte`
- `short : Short`
- `int : Integer`
- `long : Long`
- `float : Float`
- `double : Double`
- `char : Character`
- `boolean : Boolean`

The main incentives of using such wrappers in your code, are:

- Accessing type information about the corresponding primitive type
- Auto-Boxing feature, where a primitive data is automatically promoted to an object reference type
- Moving primitive type data around data structures (called *collections*), using their wrapper-style counterparts

Let's look at these incentives in the next step.

Summary

In this step, we:

- Were introduced to the wrapper classes available for the primitive types
- Learned about the advantages of having them around

Step 09: Creating Wrapper Objects

Now that we know what wrapper classes are, and which ones correspond to each primitive Java type, let's try them out.

Creating instance of Wrapper Classes using `new` is simple. Examples below.

```
Integer hundred = new Integer("100");
Boolean b1 = new Boolean("true"); //true
Boolean b1 = new Boolean("True"); //true
Boolean b1 = new Boolean("false"); //false
Boolean b1 = new Boolean("False"); //false
Boolean b1 = new Boolean("other arbitrary string"); //false
```

You can also use `valueOf()` method within types such as `Integer` and `Float` to create a wrapper object.

```
Float floatWrapper = Float.valueOf(57.0f);
int floatToInt = floatWrapper.intValue(); // 57
Integer seven = Integer.valueOf("111", 2);
Integer.toString(seven, 2);
```

Difference between creating wrapper objects using valueOf and new

The `Integer.valueOf()` reuses existing `Integer` objects with same value on the heap. If an object with same value is present in the heap, it returns a reference to existing object. Otherwise, it returns a reference of a newly created `Integer` object.

Wrapper classes are immutable. Hence, above approach is efficient and accurate.

```
jshell> Integer integer1 = new Integer(5);
integer1 ==> 5
jshell> Integer integer2 = new Integer(5);
integer2 ==> 5
jshell> Integer integer3 = Integer.valueOf(5);
integer3 ==> 5
jshell> Integer integer4 = Integer.valueOf(5);
integer4 ==> 5
jshell> integer1 == integer2
$1 ==> true
jshell> integer3 == integer4
$2 ==> true
jshell>
```

Summary

In this step, we:

- Discovered that there are two ways to create a wrapper object for primitive data
- Learned that `valueOf()` takes advantage of immutability, to improve efficiency

Step 10: Auto-Boxing, And Some Wrapper Constants

Auto-Boxing is an example of **syntactic sugar** in the Java language. It does not provide new features but makes code more readable.

Auto-boxing reuses the mechanism provided by `Integer.valueOf()` for `Integer` , `Float` and others.

```
jshell> Integer seven = Integer.valueOf(7000);
seven ==> 7000
jshell> Integer sevenToo = 7000;
sevenToo ==> 7000
jshell> Integer sevenAgain = 7000;
sevenAgain ==> 7000
jshell> sevenToo == sevenAgain
$1 ==> true
```

In the above example, `Integer sevenToo = 7000` uses auto boxing. We are upgrading a `int` literal to a `Integer` value. This is done implicitly by using `Integer.valueOf` method.

There are constants available on Wrapper classes print the size of their variables and the range of values they can store.

```
jshell> Integer.MAX_VALUE
$2 ==> 2147483647
jshell> Integer.MIN_VALUE
$3 ==> -2147483648
jshell> Integer.SIZE
$4 ==> 32
jshell> Integer.BYTES
$5 ==> 4
jshell>
```

Summary

In this step, we:

- Understood the mechanism of auto-boxing, which uses the assignment operator route
- Understood how auto-boxing internally makes use of `valueOf()` method

Step 11: The Java `Date` API

No discussion on the built-in Java primitive and reference types is complete without an exploration of the Date API.

Before Java SE 8, there were a lot of practical issues regarding the interface and implementation of the `Date` class.

From Java 8, Java provided an API for `Date` classes based on the Joda Date Framework.

The three most significant classes within this framework are : `LocalDate` , `LocalTime` and `LocalDateTime` .

Snippet-01 : java.time utilities

`java.time.*` is not imported automatically by Jshell. Let's import it in.

The commonly used utilities to access current values of date and time are:

- `LocalDate.now()` : Returns the current date value in readable format
- `LocalTime.now()` : Returns the current time value in a readable format
- `LocalDateTime.now()` : Returns a combination of the current date and time values, in a readable format

```
jshell> import java.time.LocalDate
jshell> LocalDate now = LocalDate.now()
now ==> 2018-02-01
jshell> import java.time.*
jshell> LocalDateTime now = LocalDateTime.now()
now ==> 2018-02-01T15:50:48.423164
jshell> LocalTime now = LocalTime.now()
now ==> 15:51:09.642001
jshell>
```

Summary

In this step, we:

- Were introduced to the Java Date API, available through the `java.time` package
- Saw how to use the `LocalDate` , `LocalTime` and `LocalDateTime` types

Step 12: Playing With `java.time.LocalDate`

We've been looking at calendars right from childhood, haven't we! How about playing around with a digital calendar?

`LocalDate` provides a number of methods to retrieve

- Date information: Day/Month/Year and related attributes
- Date meta-information: Classification-related information, such as whether a leap year, etc.

```
jshell> import java.time.*
jshell> import java.time.*
jshell> LocalDate today = LocalDate.now()
today ==> 2018-02-01
jshell> today.getYear()
$1 ==> 2018
jshell> today.getDayOfWeek()
$2 ==> THURSDAY
jshell> today.getDayOfMonth()
$3 ==> 01
jshell> today.getDayOfYear()
$4 ==> 32
jshell> today.getMonth()
$5 ==> FEBRUARY
jshell> today.getMonthValue()
$6 ==> 2
jshell> today.isLeapYear()
$7 ==> false
jshell> today.lengthOfYear()
$8 ==> 365
jshell> today.lengthOfMonth()
$9 ==> 28
```

`LocalDate` is also immutable. You can use different methods provided to add and subtract days, months and years. Each of these return a new instance of `LocalDate` .

```
jshell> today.plusDays(100)
$10 ==> 2018-05-12
jshell> today.plusMonths(100)
$11 ==> 2026-06-01
jshell> today.plusYears(100)
$12 ==> 2118-02-01
jshell> today.minusYears(100)
$13 ==> 1918-02-01
jshell> LocalDate yesteryear = today.minusYears(100)
yesterYear ==> 1918-02-01
```

```
jshell> today
today ==> 2018-02-01
jshell>
```

`LocalDateTime` extends `LocalDate` and provides time component as well. You can access, and perform arithmetic on the hours, minutes, seconds and nanoseconds values.

Summary

In this step, we:

- Saw common utilities that the `LocalDate` `class` provides
- Learned that `LocalDate` , `LocalTime` and `LocalDateTime` are all immutable

Step 13: Comparing `LocalDate` Objects

You can take a look at additional utility methods in `LocalDate` in examples below:

```
jshell> LocalDate today = LocalDate.now()
today ==> 2018-02-01
jshell> LocalDate yesterday = LocalDate.of(2018, 01, 31)
yesterday ==> 2018-01-31
jshell> today.withYear(2016)
$1 ==> 2016-02-01
jshell> today.withDayOfMonth(20)
$2 ==> 2018-02-20
jshell> today.withMonth(3)
$3 ==> 2018-03-01
jshell> today.withDayOfYear(3)
$4 ==> 2018-04-30
```

You can also compare dates using the methods shown below:

```
jshell> today.isBefore(yesterday)
$5 ==> false
jshell> today.isAfter(yesterday)
$6 ==> true
jshell>
```

These methods are also available with `LocalTime` and `LocalDateTime` classes as well.

Arrays and ArrayList

We will use the following exercise to understand heavily used data structures of Java - Arrays and `ArrayList` .

We would like to model a student report card in a Java program, which allows the user to do stuff such as:

```
Student student - new Student(name, list-of-marks);
int number = student.getNumberOfmarks();
int sum = student.getTotalSumOfMarks();
int maximumMark = student.getMaximumMark();
int minimumMark = student.getMinimumMark();
BigDecimal average = student.getAverageMarks();
student.addMark(35);
student.removeMarkAtIndex(5);
```

A data structure like array and `ArrayList` allow you to store multiple object of the same kind. These are called **aggregates**.

Step 01: The Need For `Array`

Let's start with understanding the need for arrays.

Consider the example below. We are creating 3 marks and adding them.

```
        jshell> int mark1;
        mark1 ==> 0
        jshell> mark1 = 100;
        mark1 ==> 10
        jshell> int mark2 = 75;
        mark2 ==> 75
        jshell> int mark3 = 60;
        mark3 ==> 60
        jshell> int sum = mark1 + mark2 + mark3;
        sum ==> 235
        jshell> int mark4 =7 56;
        mark4 ==> 56
        jshell> sum = mark1 + mark2 + mark3 + mark4;
        sum ==> 291
        jshell>
```

If an additional mark component `mark4` is added to the existing list of marks `mark1` , `mark2` and `mark3` , the code for computing `sum` needs to change.

All these marks are similar. How about creating a group of marks and storing it as part of single variable?

Let's create an array - `marks`

```
        jshell> int[] marks = {75, 60, 56};
        marks ==> int[3]{ 75,60,56 }
```

In above example

- `marks` is an `array` .
- `marks` stores multiple `int` values.
- `marks` array has 3 values

In an array, the index runs from 0 to (length of array - 1). In above example, the index runs from 0 to 2.

You can use an index to find the specific element from the array. It is done by using the indexing operator, ' `[]` '. The expression `marks[0]` maps to the first array element stored at index `0` of the array `marks` .

```
    jshell> marks[0]
    $20 ==> 75

    jshell> marks[1]
    $21 ==> 60

    jshell> marks[2]
    $22 ==> 56
```

How do we write code to sum all values in marks array?

```
jshell> int sum = 0;
sum ==> 0
jshell> for(int mark:marks) {
   ..>> sum = sum + mark;
   ..>> }
jshell> sum
sum ==> 191
jshell>
```

Above construct is called Enhanced `for` loop.

`mark` is the loop control variable. Its type needs to match the type of an array elements, which is `int`.

Above `for` loop can be used irrespective of the number of elements in `marks` array.

Step 02: Storing And Accessing Array Values

Let's dig deeper into arrays in this step.

An array can be used to store zero or more number of elements.

```
jshell> int[] marks = {1, 2, 3 };
marks ==> int[3]{ 1,2,3 }
jshell> int[] marks = {1, 2, 3, 4, 5};
marks ==> int[5]{ 1,2,3,4,5 }
jshell> int[] marks = {1};
marks ==> int[1]{ 1 }
jshell> int[] marks = {};
marks ==> int[0]{  }
```

An array can also be created with `new` operator. You've to specify the size of the array.

```
jshell> int[] marks2 = new int[5];
marks2 ==> int[5]{ 0,0,0,0,0 }
```

You can use array index to assign values.

`marks2[0] = 10` stores `10` as first element in marks array.

```
jshell> marks2[0]
$1 ==> 0
jshell> marks2[0] = 10;
$2 ==> 10
jshell> marks2[0]
$3 ==> 10
jshell>
```

Snippet below shows more examples.

```
jshell> int[] marks2 = new int[5];
marks2 ==> int[5]{ 0,0,0,0,0 }
jshell> marks2[0] = 1;
$1 ==> 1
jshell> marks2[1] = 2;
$2 ==> 2
jshell> marks2[2] = 3;
$3 ==> 3
```

```
jshell> marks2[3] = 4;
$4 ==> 4
jshell> marks2[4] = 5;
$5 ==> 5
jshell> marks2
marks2 ==> int[5]{ 1,2,3,4,5 }
```

`length` can be used to find the number of elements in the array.

```
jshell> marks2.length
$6 ==> 5
jshell> int marks3 = {};
marks3 ==> int[0]{  }
jshell> marks3.length
$7 ==> 0
```

Classroom Exercise CE-AA-01

1. Write a program that creates an array `marks` to store `8` `int` values, and code to iterate through `marks` using a `for` loop, printing out its values.

Hint: Use the `marks.length` property

Solution To CE-AA-01

```
jshell> int[] marks = {1, 2, 3, 4, 5, 6, 7, 8};
marks ==> int[8]{ 1,2,3,4,5,6,7,8 }
jshell> marks.length
$1 ==> 8
jshell> for(int i=0; i < marks.length; i++) {
   ..>> System.out.println(marks[i]);
   ..>> }
1
2
3
4
5
6
7
8
jshell>
```

Step 04: Array Initialization, Data Types And Exceptions

Below snippet shows how arrays with different types are initialized.

Summary - int - 0, double - 0.0, boolean - false, Any object - null

```
jshell> int[] marks = new int[5];
marks ==> int[5]{ 0,0,0,0,0 }
jshell> double[] values = new double[5];
values ==> double[5]{ 0.0,0.0,0.0,0.0,0.0 }
jshell> boolean[] tests = new boolean[5];
tests ==> boolean[5]{ false,false,false,false,false }
jshell> class Person {
   ..>>}
| created class Person
jshell> Person[] persons = new Person[5];
persons ==> Person[5]{ null,null,null,null,null }
jshell>
```

Let's look at a few variations of array declarations.

Important things to Remember

- The declaration part of an array definition must not include the dimension of the array
- The assignment part of an array definition must include the dimension of the array
- All the elements of an array must be of the same, homogeneous type

```
jshell> int[5] marks2;
| Error:
| ']' expected
| int[5] marks2;
|____^
jshell> int[] marks2 = new int[];
| Error:
| array dimension missing
| int[] marks2 = new int[];
|____________^========^
jshell> int[] marks2 = new int[5];
marks2 ==> int[5]{ 0,0,0,0,0 }
jshell> marks2[6]
| java.lang.ArrayIndexOutOfBoundsException thrown : 6
| at (#54:1)
jshell> int[] marks3 = {1, 2, 3, 4.0};
| Error:
| incompatible types: possible lossy conversion from double to int
| int[] marks3 = {1, 2, 3, 4.0};
|__________________________^_^
jshell>
```

The name of an array is a reference variable that stores the address of where the array is created on the heap.

```
jshell> int[] marks4 = {1, 2, 3, 4, 5};
marks4 ==> int[5]{ 1,2,3,4,5 }
jshell> System.out.println(marks4);
I@6c49835d
```

The built-in method `Arrays.toString` can be used to print out elements of an array.

```
jshell> System.out.println(Arrays.toString(marks));
[1, 2, 3, 4, 5]
jshell>
```

Step 05: Array Utilities

Let's look at a few examples for

- Iterating through An Array
- Bulk-modification of array elements
- Comparing Arrays
- Sorting Arrays

Snippet-01 : Iterating through an array

Using an enhanced `for` loop is easy and intutive.

```
jshell> int[] marks = {100, 99, 95, 96, 100};
marks ==> int[5]{ 100,99,95,96,100 }
jshell> for(int mark:marks) {
..>> System.out.println(mark);
..>> }
100
99
95
96
100
jshell> for(int i=0; i < marks.length; i++) {
..>> System.out.println(marks[i]);
..>> }
100
99
95
96
100
jshell>
```

Snippet-02 : Filling & Comparing Arrays

`Array.fill` fills the entire array with a specified value.

```
jshell> int[] marks = new int[5];
marks ==> int[5]{ 0,0,0,0,0 }
jshell> Arrays.fill(marks, 100);
jshell> marks
marks ==> int[5]{ 100,100,100,100,100 }
```

`Array.equals` compares two given arrays, and returns a `boolean` value of `true` only if

- Both arrays are of same length and
- Elements at each corresponding index are equal, for all indexes

```
jshell> int[] array1 = {1, 2, 3};
array1 ==> int[3]{ 1,2,3 }
jshell> int[] array2 = {1, 2, 3};
array2 ==> int[3]{ 1,2,3 }

jshell> Arrays.equals(array1, array2);
$1 ==> true

jshell> int[] array3 = {3, 2, 3};
array3 ==> int[3]{ 3,2,3 }
jshell> Arrays.equals(array1, array3);
$2 ==> false

jshell> int[] array4 = {1, 2};
array4 ==> int[2]{ 1,2 }
jshell> Arrays.equals(array1, array4);
$3 ==> false
```

`Array.sort` : Performs an in-position sorting of elements by comparing pairs of them at a time.

```
jshell> Arrays.sort(array3);
jshell> array3
array3 ==> int[3]{ 2,3,3 }
jshell>
```

Step 06: Classroom Exercise CE-AA-02

Exercise

Armed with the knowledge of Java arrays and their in-built utility methods, let's now solve the challenge we started off with.

```
Student student = new Student(name, list-of-marks);
int number = student.getNumberOfmarks();
int sum = student.getTotalSumOfMarks();
int maximumMark = student.getMaximumMark();
int minimumMark = student.getMinimumMark();
BigDecimal average = student.getAverageMarks();
```

We can implement the aggregate `list-of-marks` as an array.

Solution To CE-AA-02

StudentRunner.java

```
package com.in28minutes.arrays;

public class StudentRunner {
    public static void main(String[] args) {
        int[] marks = {99, 98, 100};
        Student student = new Student("Ranga", marks);

        int number = student.getNumberOfmarks();
        System.out.println("Number of marks : " + number);
        int sum = student.getTotalSumOfMarks();
        System.out.println("Sum of marks : " + sum);

        int maximumMark = student.getMaximumMark();
        System.out.println("Maximum of marks : " + maximumMark);
        int minimumMark = student.getMinimumMark();
        System.out.println("Minimum of marks : " + minimumMark);

        BigDecimal average = student.getAverageMarks();
        System.out.println("Average of marks : " + average);
        student.addMark(35);
        student.removeMarkAtIndex(5);
    }
}
```

Student.java

```
package com.in28minutes.arrays;
import java.math.BigDecimal;

public class Student {
    private String name;
    private int[] marks;

    public Student(String name, int[] marks) {
        this.name = name;
        this.marks = marks;
    }

    public int getNumberOfMarks() {
        return marks.length;
    }

    public int getTotalSumOfMarks() {
        int sum = 0;
        for(int mark:marks) {
            sum += mark;
        }
        return sum;
    }

    public BigDecimal getAverageOfMarks() {
        int sum = getTotalSumOfMarks();
        BigDecimal average = new BigDecimal(sum).divide(new BigDecimal(marks.length), 3, Roundin
    }

    public int getMaximumMark() {
        int max = Integer.MIN_VALUE;
        for(int mark:marks) {
            if(mark > max) {
                max = mark;
            }
        }
        return max;
    }

    public int getMinimumMark() {
        int min = Integer.MAX_VALUE;
        for(int mark:marks) {
            if(mark < min) {
                min = mark;
            }
        }
        return min;
    }
}
```

Step 08: Variable Arguments - The Basics

What if you want to create a method which can accept variable number of arguments?

Let's look at an example. The critical part is the parameter `int... values`.

```
    jshell> class Something {
       ..>> public void doSomething(int... values) {
       ..>> System.out.println(Arrays.toString(values));
       ..>> }
       ..>> };
    | created class Something
```

A typical parameter would've been `int values`. This allows us to pass one parameter to the method.

What difference does the three dots `...` make in `int... values`?

```
    jshell> Something thing = new Something();
    thing ==> Something@2e465f6a
    jshell> thing.doSomething(1);
    [1]
    jshell> thing.doSomething(1, 2);
    [1, 2]
    jshell> thing.doSomething(1, 2, 3);
    [1, 2, 3]
    jshell>
```

You can see that `doSomething` can be called with one, two and three parameters.

Let's look at another example:

```
jshell> void sum(int... values) {
   ..>>   int sum = 0;
   ..>>   for(value: values) {
   ..>>     sum += value;
   ..>>   }
   ..>>   System.out.println(sum);
   ..>> }
| created method sum(int...)
```

We created a `sum` method with a variable argument. Let's look at how to use it.

```
jshell> sum(1, 2)
3
jshell> sum(1, 2, 3)
6
jshell> sum(1, 2, 3, 4)
1
jshell> sum(1, 2, 3, 4, 5, 6)
21
jshell>
```

In this step, we took our first look at variable arguments. Variable arguments allow us to pass variable number of arguments to a method.

Step 09: Variable Argument Methods For `Student`

Let's add a few methods to the `Student` class to accept variable arguments.

Student.java

```
package com.in28minutes.arrays;
import java.math.BigDecimal;

public class Student {
    private String name;
    private int[] marks;

    public Student(String name, int...  marks) {
        this.name = name;
        this.marks = marks;
    }

    public int getNumberOfMarks() {
        return marks.length;
    }

    public int getTotalSumOfMarks() {
        int sum = 0;
        for(int mark:marks) {
            sum += mark;
        }
        return sum;
    }

    public BigDecimal getAverageOfMarks() {
        int sum = getTotalSumOfMarks();
        BigDecimal average = new BigDecimal(sum).divide(new BigDecimal(marks.length), 3, Roundin
    }

    public int getMaximumMark() {
        int max = Integer.MIN_VALUE;
        for(int mark:marks) {
            if(mark > max) {
                max = mark;
            }
        }
        return max;
    }

    public int getMinimumMark() {
        int min = Integer.MAX_VALUE;
        for(int mark:marks) {
            if(mark < min) {
                min = mark;
            }
        }
        return min;
    }
}
```

StudentRunner.java

```
package com.in28minutes.arrays;
public class StudentRunner {
    public static void main(String[] args) {
        //int[] marks = {99, 98, 100};
        Student student = new Student("Ranga", 97, 98, 100);

        int number = student.getNumberOfmarks();
        System.out.println("Number of marks : " + number);
        int sum = student.getTotalSumOfMarks();
        System.out.println("Sum of marks : " + sum);

        int maximumMark = student.getMaximumMark();
        System.out.println("Maximum of marks : " + maximumMark);
        int minimumMark = student.getMinimumMark();
        System.out.println("Minimum of marks : " + minimumMark);
        BigDecimal average = student.getAverageMarks();
        System.out.println("Average of marks : " + average);

        student.addMark(35);
        student.removeMarkAtIndex(5);
    }
}
```

Quick Tip

The variable arguments list must always be at the end of the parameter list passed to a method. The following method definition **will not** be accepted by the Java compiler:

```
void process(int... values, String name) {

}
```

Step 10: Arrays - Some Puzzles And Exercises

Let's look at a few puzzles and exercises with Arrays.

When creating arrays of objects, the array ends up holding references to the created objects.

```
jshell> class Person {
   ..>> }
| created class Person
jshell> Person[] persons = new Person[5];
persons ==> Person[5]{ null,null,null,null,null }
```

We can assign new `Person` objects to different elements of the array.

```
jshell> persons[1] = new Person();
$1 ==> Person@394e1a0f
jshell> persons
persons ==> Person[5]{ null,Person@394e1a0f,null,null,null }
jshell> persons[0] = new Person();
$2 ==> Person@1e965684
jshell> persons
persons ==> Person[5]{ Person@1e965684,Person@394e1a0f, null,null,null }
```

You can also initialize values when you create an array.

```
jshell> Person[] persons2 = {new Person(), new Person()};
persons2 ==> Person[2]{ Person@3b088d51, Person@1786dec2 }
```

Here's how you can initialize a `String` array.

```
jshell> String[] textValues = {"Apple", "Ball", "Cat"};
textValues ==> String[2]{ "Apple","Ball","Cat" }
jshell>
```

Classroom Exercise CE-AA-03

Exercises

1. Create a `String` array with the names of days of the week:

`Sunday` , `Monday` , `Tuesday` , `Wednesday` , `Thursday` , `Friday` , `Saturday` 2. Find the day with the most number of letters in it 3. Print days of the week backwards

Solutions To CE-AA-03

WeekRunner.java

```
package com.in28minutes.arrays;

public class StringRunner {

    public static void main(String[] args) {

        String[] daysOfWeek = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Sa

        String dayWithMostCharacters = "";

        for (String day : daysOfWeek) {
            if (day.length() > dayWithMostCharacters.length()) {
                dayWithMostCharacters = day;
            }
        }

        System.out.println("Day with Most number of characters " + dayWithMostCharacters);

        for (int i = daysOfWeek.length - 1; i >= 0; i--) {
            System.out.println(daysOfWeek[i]);
        }

    }

}
```

Step 11: Problems With Arrays

Let's look at our implementation for our challenge.

We've implemented most of the features except for `addMark()` and `removeMarkAtIndex()` .

```
    Student student = new Student(name, <list-of-marks>);
    int number = student.getNumberOfmarks();
    int sum = student.getTotalSumOfMarks();
    int maximumMark = student.getMaximumMark();
    int minimumMark = student.getMinimumMark();
    BigDecimal average = student.getAverageMarks();
    student.addMark(35);
    student.removeMarkAtIndex(5);
```

We would want to add and remove from an array. Can we do this?

The size of an array is fixed at its compile-time definition. Which means that once we define an array such as:

`String[] textValues = {"Apple", "Ball", "Cat"};`

The size of the `textValues` array is fixed to 3. You an change values inside the array. But the size cannot be changed.

How to add an element to an array?

One of the options is

- Create a fresh array with a few extra element slots to accommodate the additional elements to be inserted
- Copy the existing array elements to the beginning of this new array
- Add the additional elements at the rear end of this array

If elements need to be removed from an array:

- Create a fresh array with correspondingly lesser element slots
- Copy the existing array elements, excluding the ones to be removed, to the beginning of this new array

```
jshell> int[] marks = {12, 34, 45};
marks ==> int[3] { 12, 34, 45 }

jshell> int[] newMarks = new int[marks.length+1];
newMarks ==> int[4] { 0, 0, 0, 0 }

jshell> System.arraycopy(marks, 0, newMarks, 0, marks.length);

jshell> newMarks
newMarks ==> int[4] { 12, 34, 45, 0 }
```

```
jshell> newMarks[3] = 100
$27 ==> 100

jshell> newMarks
newMarks ==> int[4] { 12, 34, 45, 100 }
```

As you can see, this can be very inefficient. How do we solve it?

Step 12: Introducing `ArrayList`

`ArrayList` is more dynamic than an array. It provides operations to add and remove elements.

Let's start with creating an `ArrayList` and add a few values.

```
    jshell> ArrayList arrayList = new ArrayList();
    arrayList ==> []
    jshell> arrayList.add("Apple");
    | Warning:
    | unchecked call to add(E) as a member of the raw type java.util.ArrayList
    | arrayList.add("Apple");
    |_^--------------------^
    $1 ==> true
    jshell> arrayList.add("Ball");
    | Warning:
    | unchecked call to add(E) as a member of the raw type java.util.ArrayList
    | arrayList.add("Ball");
    |_^--------------------^
    $2 ==> true
    jshell> arrayList.add("Cat");
    | Warning:
    | unchecked call to add(E) as a member of the raw type java.util.ArrayList
    | arrayList.add("Cat");
    |_^--------------------^
    $3 ==> true
    jshell> arrayList
    arrayList ==> ["Apple", "Ball", "Cat"]
```

You can remove values using `remove` method.

```
    jshell> arrayList.remove("Cat");
    $4 ==> true
    jshell> arrayList
    arrayList ==> ["Apple", "Ball"]
```

The `ArrayList` instance `arrayList` can be used to store objects of pretty much any type, even primitive types. Also, non-homogeneous types!

> The warning message displayed is a hint to the programmer to discourage this.

```
    jshell> arrayList.add(1);
    | Warning:
    | unchecked call to add(E) as a member of the raw type java.util.ArrayList
    | arrayList.add(1);
    |_^--------------------^
    $3 ==> true
    jshell> arrayList
    arrayList ==> ["Apple", "Ball", 1]
    jshell>
```

Let's say we want to store only `String` values in an `ArrayList`. How do we do it?

We can specify the type of elements that an `ArrayList` can contain.

```
jshell> ArrayList<String> items = new ArrayList<String>();
items ==> []
```

In above snippet, we are creating an `ArrayList` that can hold `String` values.

You can add `String` value but not numbers.

```
jshell> items.add("Apple");
$1 ==> true
jshell> items
items ==> ["Apple"]
jshell> items.add(1);
| Error
| no suitable method found for add(int)
|...
jshell> items.add("Ball");
$2 ==> true
jshell> items.add("Cat");
$3 ==> true
jshell> items
items ==> ["Apple", "Ball", "Cat"]
```

Rest of operations are similar to a normal `ArrayList`.

```
jshell> items.remove("Cat");
$4 ==> true
jshell> items
items ==> ["Apple", "Ball"]
jshell> items.remove(0);
$5 ==> "Apple"
jshell> items
items ==> ["Ball"]
jshell>
```

Step 13: Refactoring `Student` To Use `ArrayList`

Let's now get to the `Student` challenge once again. Let's use an `ArrayList` this time.

```
Student student = new Student(name, <list-of-marks>);
int number = student.getNumberOfmarks();
int sum = student.getTotalSumOfMarks();
int maximumMark = student.getMaximumMark();
int minimumMark = student.getMinimumMark();
BigDecimal average = student.getAverageMarks();
```

Snippet-01 : Refactoring `Student`

StudentRunner.java

```
package com.in28minutes.arrays;

public class StudentRunner {
    public static void main(String[] args) {
        Student student = new Student("Ranga", 97, 98, 100);
        int number = student.getNumberOfmarks();
        System.out.println("Number of marks : " + number);
        int sum = student.getTotalSumOfMarks();
        System.out.println("Sum of marks : " + sum);

        int maximumMark = student.getMaximumMark();
        System.out.println("Maximum of marks : " + maximumMark);
        int minimumMark = student.getMinimumMark();
        System.out.println("Minimum of marks : " + minimumMark);

        BigDecimal average = student.getAverageMarks();
        System.out.println("Average of marks : " + average);
        System.out.println(student);
    }
}
```

Student.java

```
    package com.in28minutes.arrays;
    import java.math.BigDecimal;
    import java.math.RoundingMode;
    import java.util.ArrayList;

    public class Student {
        private String name;
        private ArrayList<Integer>  marks = new ArrayList<Integer>();

        public Student(String name, int...  marks) {
            this.name = name;
            for(int mark: marks) {
                this.marks.add(mark);
            }
        }

        public int getNumberOfMarks() {
            return marks.size();
        }

        public int getTotalSumOfMarks() {
            int sum = 0;
            for(int mark:marks) {
                sum += mark;
            }
            return sum;
        }

        public BigDecimal getAverageOfMarks() {
            int sum = getTotalSumOfMarks();
            BigDecimal average = new BigDecimal(sum).divide(new BigDecimal(marks.size()), 3, Roundin
        }

        public int getMaximumMark() {
            return Collections.max(marks);
        }

        public int getMinimumMark() {
            return Collections.min(marks);
        }

        public String toString() {
            return name + marks;
        }
    }
```

The Enhanced `for` loop works for `ArrayList` s as well, just like in the case of an array

The `Collections.max()` and `Collections.min()` methods can be used to find the maximum and minimum value in an array.

Step 14: Enhancing `Student` Further

Let's now add the features to add and remove a student.

```
package com.in28minutes.arrays;

public class StudentRunner {
    public static void main(String[] args) {
        Student student = new Student("Ranga", 97, 98, 100);
        int number = student.getNumberOfmarks();
        System.out.println("Number of marks : " + number);
        int sum = student.getTotalSumOfMarks();
        System.out.println("Sum of marks : " + sum);

        int maximumMark = student.getMaximumMark();
        System.out.println("Maximum of marks : " + maximumMark);
        int minimumMark = student.getMinimumMark();
        System.out.println("Minimum of marks : " + minimumMark);

        BigDecimal average = student.getAverageMarks();
        System.out.println("Average of marks : " + average);
        System.out.println(student);

        student.addMark(35);
        System.out.println(student);
        student.removeMarkAtIndex(1);
        System.out.println(student);
    }
}
```

Student.java

```
package com.in28minutes.arrays;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.ArrayList;

public class Student {
    private String name;
    private ArrayList<Integer>  marks = new ArrayList<Integer>();

    public Student(String name, int...  marks) {
        this.name = name;
        for(int mark: marks) {
            this.marks.add(mark);
        }
    }

    public int getNumberOfMarks() {
        return marks.size();
    }

    public int getTotalSumOfMarks() {
        int sum = 0;
        for(int mark:marks) {
            sum += mark;
        }
        return sum;
    }

    public BigDecimal getAverageOfMarks() {
        int sum = getTotalSumOfMarks();
        BigDecimal average = new BigDecimal(sum).divide(new BigDecimal(marks.size()), 3, Rounding
    }

    public int getMaximumMark() {
        return Collections.max(marks);
    }

    public int getMinimumMark() {
        return Collections.min(marks);
    }

    public String toString() {
        return name + marks;
    }

    public void addMark(int mark) {
        marks.add(mark);
    }

    public void removeMarkAtIndex(int index) {
        marks.remove(index);
    }
}
```

Console Output

Number of marks : 3

Sum of marks : 3

Average of marks : 3_

Maximum of marks : 3

Minimum of marks : 3

Ranga[97,98,100]

Ranga[97,98,100,35]

Ranga[97,100,35]

Object Oriented Programming (*OOP*) - Revisited

In this section, we revisit the principles of *OOP*, armed with the knowledge of

- Arrays and their variants
- Built-in Java classes and utilities, and
- Conditionals and loops (normal and enhanced).

Step 01: Objects Revisited - State And Behavior

The attributes of an object determine what it is made up of. At different points in an object's lifetime, the value of any of its attributes can change.

At any given time, values of these attributes defines the object's **state**.

In The `MotorBike` example, the attribute `speed` defines a `MotorBike` 's state. The `speed` of a `ducati` defines its state.

How an object responds to an external event, or a message sent to it, defines its **behavior**.

Messages are delivered to an object using methods. Methods are used to implement an object's **behavior**.

The methods `setSpeed` , `increaseSpeed` and `decreaseSpeed` have an effect on the observed speed of the `MotorBike` s. The future `state` of a `MotorBike` depends on `behavior` and current `state` .

`behavior` affects `state` . And `state` affects `behavior` .

MotorBikeRunner.java

```
package com.in28minutes.oops;

public class MotorBikeRunner {
    public static void main(String[] args) {
        MotorBike ducati = new MotorBike(100);
        MotorBike honda = new MotorBike(200);
        MotorBike yamaha = new MotorBike();
        ducati.start();
        honda.start();
        yamaha.start();

        ystem.out.println(ducati.getSpeed());
        System.out.println(honda.getSpeed());
        System.out.println(yamaha.getSpeed());

        ducati.increseSpeed(50);
        yamaha.setSpeed(250);
        honda.increaseSpeed(100);
        yamaha.decreaseSpeed(50);
        System.out.println(ducati.getSpeed());
        System.out.println(honda.getSpeed());
        System.out.println(yamaha.getSpeed());
    }
}
```

MotorBike.java

```
package com.in28minutes.oops;

public class MotorBike {
    //state
    private int speed;
    //behavior
    MotorBike() {
        this(5);
    }

    MotorBike(int speed) {
        if(speed > 0)
            this.speed = speed;
    }

    public void start() {
        System.out.println("Bike started!");
    }

    public void setSpeed(int speed) {
        if(speed > 0)
            this.speed = speed;
    }

    public int getSpeed() {
        return this.speed;
    }

    public void increaseSpeed(int howMuch) {
        setSpeed(this.speed + howMuch);
    }

    public void decreaseSpeed(int howMuch) {
        setSpeed(this.speed - howMuch);
    }
}
```

Step 02: Managing `class` state

At a basic level, when we design a class, we decide:

- `state` - member variables
- `how to create objects` - Define constructors
- `behavior` - What methods are exposed

Consider the example of a `Fan` `class`.

The above three major areas that correspond to its design could be as follows:

- State
  - make
  - radius
  - color
  - isOn
  - speed
- Constructors
  - Fan(String make, double radius, String color)
- Behavior
  - void switchOn()
  - void SwitchOff()
  - void changeSpeed(int change)
  - String toString()

Let's try to write a simple `Fan` `class`, that covers all these aspects.

Fan.java

```
    package com.in28minutes.oops.level2;

    public class Fan {
        //state
        private String make;
        private double radius;
        private String color;
        private boolean isOn;
        private byte speed;    //levels: 0 to 5

        //constructors

        public Fan(String make, double radius, String color) {
            this.make = make;
            this.radius = radius;
            this.color = color;
        }

        //methods
        public String toString() {
            return String.format("Make : %s, Radius : %f, Color : %s, Is On : %b, Speed : %d",
                                 make,
                                 radius,
                                 color,
                                 isOn,
                                 speed);
        }
    }
```

FanRunner.java

```
package com.in28minutes.oops.level2;

public class FanRunner {
    public static void main(String[] args) {
        Fan fan = new Fan("Fan-Tastic", 0.456, "GREEN");
        System.out.println(fan);
    }
}
```

Console Output

Make : Fan-tastic, Radius : 0.45600, Color : GREEN, Is On : false, Speed : 0

The fields which were not set by the constructor, namely `isOn` and `speed` , assumed the language default values for their data types, namely `false` (for `booelan`) and `0` (for `int`).

Step 03: Augmenting `Fan` With Behavior

We need to decide what kind of behavior should be provided by a `Fan` object.

The default state attributes of the `Fan` class objects, namely `make` , `color` and `radius` are fixed at manufacturing time, and cannot be altered by a user of this `class` 's instances.

The other two state attributes, `isOn` and `speed` need to be exposed to change by `Fan` object users. We will offer methods that change them.

Snippet-01 : The `Fan` `class` - v4

FanRunner.java

```
package com.in28minutes.oops.level2;

public class FanRunner {
    public static void main(String[] args) {
        Fan fan = new Fan("Fan-Tastic", 0.456, "GREEN");
        System.out.println(fan);
        fan.switchOn();
        System.out.println(fan);
        fan.setSpeed((byte)5);
        System.out.println(fan);
        fan.switchOff();
        System.out.println(fan);
    }
}
```

Fan.java

```
package com.in28minutes.oops.level2;

public class Fan {
    //state
    private String make;
    private double radius;
    private String color;
    private boolean isOn;
    private byte speed;    //levels: 0 to 5

    //constructors
    public Fan(String make, double radius, String color) {
        this.make = make;
        this.radius = radius;
        this.color = color;
    }

    //methods
    public String toString() {
        return String.format("Make : %s, Radius : %f, Color : %s, Is On : %b, Speed : %d",
                                make,
                                radius,
                                color,
                                isOn,
                                speed);
    }

    //isOn
    /*public void isOn(boolean isOn) {
        this.isOn = isOn;
    }*/

    public void switchOn() {
        isOn = true;
        setSpeed((byte)1);
    }

    public void switchOff() {
        isOn = false;
        setSpeed((byte)0);
    }

    public void setSpeed(byte speed) {
        this.speed = speed;
    }
}
```

Console Output

Make : Fan-Tastic, Radius : 0.45600, Color : GREEN, Is On : false, Speed : 0

Make : Fan-Tastic, Radius : 0.45600, Color : GREEN, Is On : true, Speed : 1

Make : Fan-Tastic, Radius : 0.45600, Color : GREEN, Is On : true, Speed : 5

Make : Fan-Tastic, Radius : 0.45600, Color : GREEN, Is On : false, Speed : 0

Snippet-01 Explained

- Regarding the state attribute `isOn` :
    - A state modifier method such as `public void isOn(boolean)` is not preferred, even though it does alter this attribute. This is because it is not intuitive from the `class` user's perspective.
    - Alternatively, methods such as `public void switchOn()` and `public void switchOff()` not only toggle the attribute `isOn` , but are also intuitive and useful to the `Fan class` users (Here, the `FanRunner` class).
- Regarding the state attribute `speed` :
    - `setSpeed` is both intuitive as well as useful, so not much rethinking needed here
    - `speed` needs to be affected by the operations `switchOn()` and `switchOff()` . We have added calls to `setSpeed()` in these method definitions.

Summary

The best way to design a class is using an `Outside In` thought process:

- Who all could possibly be using my `class` ?
- What functionality would they absolutely require?

Step 04: Programming Exercise PE-OOP-01

1. Write a simple `Rectangle class` , while covering the following constituents:

- State
    - length
    - width
- Constructors
- Behavior or Methods

Solution To PE-OOP-01

RectangleRunner.java

```java
package com.in28minutes.oops.level2;

public class RectangleRunner {
    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(12, 23);
        System.out.println(rectangle);
        rectangle.setWidth(25);
        System.out.println(rectangle);
        rectangle.setLength(20);
        System.out.println(rectangle);
    }
}
```

Rectangle.java

```java
package com.in28minutes.oops.level2;

public class Rectangle {
    //state:
        private int length;
        private int width;

    //creation:
        public Rectangle(int length, int width) {
            this.length = length;
            this.width = width;
        }

        //behaviors:
        public int getLength() {
            return length;
        }

        public int getWidth() {
            return width;
        }

        public void setLength(int length) {
            this.length = length;
        }

        public void setWidth(int width) {
            this.width = width;
        }

        public int area() {
            return length * width;
        }

        public int perimeter() {
            return 2*(length + width);
        }

        public String toString() {
            return String.format("Rectangle - length : %d, width : %d, area : %d, perimeter : %d",
                                    length,
                                    width,
                                    area(),
                                    perimeter());
        }
    }
```

Console Output

Rectangle - length : 12, width : 23, area : 276, perimeter : 70

Rectangle - length : 12, width : 25, area : 300, perimeter : 74

Rectangle - length : 20, width : 25, area : 500, perimeter : 90

Solution Explained

- A `Rectangle` object created without a specified `length` and `width` makes no practical sense, therefore a default constructor is not provided.

Step 06: Understanding Object Composition

Let's take a re-look at the state attributes of the `Fan` `class` :

Fan.java

```
    package com.in28minutes.oops.level2;

    public class Fan {
        //state
        private String make;
        private double radius;
        private String color;
        private boolean isOn;
        private byte speed;

        //constructors
        //methods
    }
```

All member variables of 'Fan' class are primitive variables. Can we make it complex and include other classes?

Snippet-01 : Object composition - State

CustomerRunner.java

```
package com.in28minutes.oops.level2;

public class CustomerRunner {
    public static void main(String[] args) {
        Customer customer = new Customer();
    }
}
```

Address.java

```
package com.in28minutes.oops.level2;

public class Address {
    //state
    private String doorNo;
    private String streetInfo;
    private String city;
    private String zipCode;

    //creation
    //behaviors
}
```

Customer.java

```
package com.in28minutes.oops.level2;

public class Customer {
    //state
    private String name;
    private Address homeAddress;
    private Address workAddress;

    //creation
    //behaviors
}
```

Snippet-01 Explained

`Customer customer` is composed of:

- `name` ,
- `homeAddress` , and
- `workAddress` .

`String` is a built-in type, and is simple. `Address` is a user defined type, and is composed of:

- `doorNo` ,
- `streetInfo` ,
- `city` , and
- `zipCode`

Snippet-02 : Object Composition v2 - Construction

Let's now add constructors to allow easy creation of these objects.

CustomerRunner.java

```java
package com.in28minutes.oops.level2;

public class CustomerRunner {
    public static void main(String[] args) {
        //Customer customer = new Customer();
        Address homeAddress = new Address("Flat No. 51", "Hiranandani Gardens", Mumbai", "400076'
        Address workAddress = new Address("Administrative Office", "Western Block", "Mumbai", "40
        Customer customer = new Customer("Ashwin Tendulkar", homeAddress, workAddress);
    }
}
```

Address.java

```java
package com.in28minutes.oops.level2;

public class Address {
    //state
    private String doorNo;
    private String streetInfo;
    private String city;
    private String zipCode;

    //creation
    public Address(String doorNo, String streetInfo, String city, String zipCode) {
        this.doorNo = doorNo;
        this.streetInfo = streetInfo;
        this.city = city;
        this.zipCode = zipCode;
    }

    //behaviors
}
```

Customer.java

```java
package com.in28minutes.oops.level2;

public class Customer {
    //state
    private String name;
    private Address homeAddress;
    private Address workAddress;

    //creation
    //workAddress not mandatory for creation
    public Customer(String name, String homeAddress) {
        this.name = name;
        this.homeAddress = homeAddress;
    }

    //behaviors
}
```

Snippet-9 : Object Composition v3 : Behaviors

Let's add methods to provide behavior.

CustomerRunner.java

```
    package com.in28minutes.oops.level2;

    public class CustomerRunner {
        public static void main(String[] args) {
            //Customer customer = new Customer();
            Address homeAddress = new Address("Flat No. 51", "Hiranandani Gardens", "Mumbai", "40007
            Customer customer = new Customer("Ashwin Tendulkar", homeAddress);
            System.out.println(customer);
            Address workAddress = new Address("Administrative Office", "Western Block", "Mumbai", "4
            customer.setWorkAddress(workAddress);
            System.out.println(customer);
        }
    }
```

Address.java

```
    package com.in28minutes.oops.level2;

    public class Address {
        //state
        private String doorNo;
        private String streetInfo;
        private String city;
        private String zipCode;

        //creation
        public Address(String doorNo, String streetInfo, String city, String zipCode) {
            super();
            this.doorNo = doorNo;
            this.streetInfo = streetInfo;
            this.city = city;
            this.zipCode = zipCode;
        }

        //behaviors
        public String toString() {
            return doorNo + ", " + streetInfo + ", " + city + " - " + zipCode;
        }
    }
```

Customer.java

```
    package com.in28minutes.oops.level2;

    public class Customer {
        //state
        private String name;
        private Address homeAddress;
        private Address workAddress;

        //creation
        //workAddress not mandatory for creation
        public Customer(String name, String homeAddress) {
            this.name = name;
            this.homeAddress = homeAddress;
        }

        //behaviors
        //certain components of homeAddress and workAddress can be modified, not the name
        public void setHomeAddress(Address homeAddress) {
            this.homeAddress = homeAddress;
        }

        public void setWorkAddress(Address workAddress) {
            this.workAddress = workAddress;
        }

        public Address getHomeAddress() {
            return homeAddress;
        }

        public Address getWorkAddress() {
            return workAddress;
        }

        public String toString() {
            return String.format("Customer [%s] lives at [%s], works at [%s]",
                                    name,
                                    homeAddress,
                                    workAddress);
        }
    }
```

Console Output

Customer [Ashwin Tendulkar] lives at [Flat No. 51, Hiranandani Gardens, Mumbai - 400076], works at [null]

Customer [Ashwin Tendulkar] lives at [Flat No. 51, Hiranandani Gardens, Mumbai - 400076], works at [Administrative Office, Western Block, Mumbai - 400076]

Step 07: Programming Exercise PE-OOP-02

Exercises

Write a program that manages Books and their Reviews:

- Book:
  - Id
  - Name
  - Author
- Review:
  - Id
  - Description
  - Rating

```
    Book book = new Book(123, "Object Oriented Programming With Java", "Ranga");
    book.addReview(new Review(10, "Great Book", 4));
    book.addReview(new Review(101, "Awesome", 5));
    System.out.println(book);
```

Solution To PE-OOP-02

BookReviewRunner.java

```java
package com.in28minutes.oops.level2;

public class BookReviewRunner {
    public static void main(String[] args) {
        Book book = new Book(123, "Object Oriented Programming With Java", "Ranga");
        book.addReview(new Review(10, "Great Book", 4));
        book.addReview(new Review(101, "Awesome", 5));
        System.out.println(book);
    }
}
```

Review.java

```java
package com.in28minutes.oops.level2;

public class Review {
    private int id;
    private String description;
    private byte rating;

    public Review(int id, String description, byte rating) {
        this.id = id;
        this.description = description;
        this.rating = rating;
    }

    public String toString() {
        return "(Review-" + id + ", " + description + ", " + rating + ")";
    }
}
```

Book.java

```java
package com.in28minutes.oops.level2;

public class Book {
    private int id;
    private String title;
    private String author;

    private ArrayList<Review> reviewList = new ArrayList<Review>();
    public Book(int id, String title, String author) {
        this.id = id;
        this.title = title;
        this.author = author;
    }

    public void addReview(Review review) {
        reviewList.add(review);
    }

    public String toString() {
        return "Book-" + id + ",  " + title + ", " + author + ", " + reviews);
    }
}
```

Console Output

Book-123, Object Oriented Programming With Java, Ranga, [(Review-10, Great Book", 4), (Review-101, Awesome, 5)]

Step 07: The Need For Inheritance

Let's look at two classes `Person` and `Student` .

Person.java

```
package com.in28minutes.oops.level2.inheritance;

public class Person {
    private String name;
    private String email;
    private String phoneNumber;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }
}
```

StudentWithoutInheritance.java

```
package com.in28minutes.oops.level2.inheritance;

public class StudentWithoutInheritance {
    private String name;
    private String email;
    private String phoneNumber;
    private String college;
    private int year;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setCollege(String college) {
        this.college = college;
    }

    public String getCollege() {
        return college;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public int getYear() {
        return year;
    }
}
```

In above code examples, you can see that there is a lot of

- The member fields of `Person`, namely `name`, `email` and `phoneNumber`, are replicated in `Student`.
- The setter and getter methods pertaining to the above fields of `Person` get repliated in `Student` as well.

Every `Student` is a `Person`. What if we could extend `Person` class instead of duplicating everything?

Enter Inheritance

`Student` **is a** `Person`. Java supports one of the basic Object Oriented Programming Paradigms : **Inheritance**.

`Student` can *inherit* from `Person`, to model the fact that a `Student` *is a* `Person`.

This is accomplished by using the Java keyword `extends`, during class definition of `Student`.

```
    public class Person {
        // <Person Definition>
    }

    public class Student extends Person {
        // <Student Definition, after reusing Person Code>
    }
```

Inheritance is a mechanism of code reuse. In this case, all the fields and methods previously defined in `Person` are available for `Student` as well.

In this Inheritance relationship, `Person` is called the **super-class** of `Student`. Likewise, `Student` is the **sub-class** of `Person`.

Let's now look at how we go about changing the `Student` `class` definition.

Snippet-02 : Student inherits from Person - v1

Person.java

```java
package com.in28minutes.oops.level2.inheritance;

public class Person {
    private String name;
    private String email;
    private String phoneNumber;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }
}
```

Student.java

```java
package com.in28minutes.oops.level2.inheritance;

public class Student extends Person {
    private String collegeName;
    private int year;

    public void setCollegeName(String college) {
        this.collegeName = collegeName;
    }

    public String getCollegeName() {
        return collegeName;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public int getYear() {
        return year;
    }
}
```

StudentRunner.java

```java
package com.in28minutes.oops.level2.inheritance;
public class StudentRunner {
    public static void main(String[] args)
        Student student = new Student();
        // < all setter() and getter() methods of Person and Student available >
        student.setName("Ranga");
        student.setEmail("in28minutes@gmail.com");
    }
}
```

Step 09: Introducing `Object` class

In the Java language, every class, whether an in-built Java library class, or a user-defined class, implicitly inherits from the class `Object` .

This `Object` class is available in the Java system package `java.lang` . This class is at the root of the Java class hierarchy. All classes, including arrays, implement/inherit the methods of this `class` .

Let's take a look at the `Person` and `Student` classes.

Snippet-01 : The Object class

Person.java

```java
package com.in28minutes.oops.level2.inheritance;

public class Person {
    private String name;
    private String email;
    private String phoneNumber;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }
}
```

Student.java

```java
package com.in28minutes.oops.level2.inheritance;
```

```
public class Student extends Person {
    private String collegeName;
    private int year;

    public void setCollegeName(String college) {
        this.collegeName = collegeName;
    }

    public String getCollegeName() {
        return collegeName;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public int getYear() {
        return year;
    }
}
```

StudentRunner.java

```
package com.in28minutes.oops.level2.inheritance;

public class StudentRunner {
    public static void main(String[] args)
        //Student student = new Student();
        //student.setName("Ranga");
        //student.setEmail("in28minutes@gmail.com");

        Person person = new Person();
        String personStr = person.toString();
        System.out.println(personStr);
        System.out.println(person);
        int hashCode = person.hashCode();
        person.notify();
    }
}
```

Console Output

com.in28minutes.oops.level2.inheritance.Person@7a46a697

com.in28minutes.oops.level2.inheritance.Person@7a46a697

Snippet-01 Explained

Methods of the `Object` `class` such as `toString()`, `hashCode()` and `notify()` are available to objects of `class` `Person` as default implementations.

The statement `System.out.println(person);` actually gets translated to `System.out.println(person.toString())`, as the Java system implicitly makes the call `person.toString()` as it is inherited from `class` `Object` for use in the `String` context.

Step 10: Inheritance And Method Overriding

Sub-class inherit features from super-class

- state attributes : super-class member variables
- behavior components : super-class method definitions

These are of course, available for access (and modification), and invocation, respectively, within the sub-class.

You can also override super class method implementations in a sub class - **method overriding**.

Snippet-01: Method Overriding

Person.java

```
package com.in28minutes.oops.level2.inheritance;

public class Person {
    private String name;
    private String email;
    private String phoneNumber;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public String toString() {
        return String.format("Person %s , Email : %s, Phone Number : %s", name, email, phoneNumbe
    }
}
```

PersonRunner.java

```
package com.in28minutes.oops.level2.inheritance;

public class PersonRunner {
    public static void main(String[] args)
        Person person = new Person();
        person.setName("Ranga");
        person.setEmail("in28minutes@gmail.com");
        person.setPhoneNumber("9898989898");
        String personStr = person.toString();
        System.out.println(personStr);
        System.out.println(person);
    }
}
```

Console Output

Person Ranga , Email : in28minutes@gmail.com, Phone Number : 9898989898

Person Ranga , Email : in28minutes@gmail.com, Phone Number : 9898989898

Snippet-01 Explained

By defining the method `toString()` within the `Person` sub- `class` , we are overriding the default version provided by the `Object` super- `class` .

Step 11: Classroom Exercise CE-OOP-01

Create an Employee class extending Student Class with following attributes:

- Title
- Employer
- EmployeeGrade
- Salary

Create a method toString() within Employee to print all state attribute values, including those of Person.

Snippet-01 : Employee Inheritance

Person.java

```
package com.in28minutes.oops.level2.inheritance;

public class Person {
    private String name;
    private String email;
    private String phoneNumber;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public String toString() {
        return Sring.format("Person %s , Email : %s, Phone Number : %s", name, email, phoneNumbe
    }
}
```

Employee.java

```java
package com.in28minutes.oops.level2.inheritance;
import java.math.BigDecimal;

public class Employee extends Person {
    private String title;
    private String employerName;
    private char employeeGrade;
    private BigDecimal salary;

    public void setTitle(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    public void setEmployerName(String employer) {
        this.employerName = employerName;
    }

    public String getEmployerName() {
        return employerName;
    }

    public void setEmployeeGrade(char  employeeGrade) {
        this.employeeGrade = employeeGrade;
    }

    public char getEmployeeGrade() {
        return employeeGrade;
    }

    public void setSalary(BigDecimal  salary) {
        this.salary = salary;
    }

    public BigDecimal getSalary() {
        return salary;
    }

    public String toString() {
        return String.format("Employee Title: %s, Employer: %s, Employee Grade: %c, Salary: %s",
                                title,
                                employerName,
                                employeeGrade,
                                salary);
    }
}
```

EmployeeRunner.java

```java
package com.in28minutes.oops.level2.inheritance;

public class EmployeeRunner {
    public static void main(String[] args) {
        Employee employee = new Employee();
        employee.setName("Ranga");
        employee.setEmail("in28minutes@gmail.com");
        employee.setPhoneNumber("123-456-7890");
        employee.setTitle("Programmer Analyst");
        employee.setEmployerName("In28Minutes");
        employee.setEmployeeGrade('A');
```

```
            employee.setSalary(new BigDecimal("50000"));
            System.out.println(employee);
        }
    }
```

Console Output

Employee Title: Programmer Analyst, Employer: In28Minutes, Employee Grade: A, Salary: 50000.0000

Snippet-01 Explained

We have not printed the underlying `Person` object details in `Employee.toString()` method overriding. Let's look to do that next.

Step 12: Constructors, And Calling `super()`

The `super` keyword allows an sub-class to access the attributes present in the super-class.

Snippet-01 : Calling Person.toString()

Employee.java

```
    package com.in28minutes.oops.level2.inheritance;
    import java.math.BigDecimal;

    public class Employee extends Person {
        //focusing only on the toString() method
        public String toString() {
            return String.format("Employee Name: %s, Email: %s, Phone Number: %s, Title: %s, Employe
                                    super.getName(),
                                    super.getEmail(),
                                    super.getPhoneNumber(),
                                    title,
                                    employerName,
                                    employeeGrade,
                                    salary);
        }
    }
```

EmployeeRunner.java

```
    package com.in28minutes.oops.level2.inheritance;

    public class EmployeeRunner {
        public static void main(String[] args) {
            Employee employee = new Employee();
            employee.setName("Ranga");
            employee.setEmail("in28minutes@gmail.com");
            employee.setPhoneNumber("123-456-7890");
            employee.setTitle("Programmer Analyst");
            employee.setEmployerName("In28Minutes");
            employee.setEmployeeGrade('A');
            employee.setSalary(new BigDecimal("50000"));
            System.out.println(employee);
        }
    }
```

Console Output

Employee Name: Ranga, Email: in28minutes@gmail.com, Phone Number: 123-456-7890, Title: Programmer Analyst, Employer: In28Minutes, Employee Grade: A, Salary: 50000.0000

The `super` keyword allows an sub-class to access the attributes present in the super-class. Hence, we were able to invoke the getter methods of the `Person` object within the `Employee` object, like this within `Employee.toString()` * `super.getName()` * `super.getEmail()` * `super.getPhoneNumber()`

Sub-Class Contructor

What happens when a sub class object is created? Does the super class constructor get called?

Snippet-7 : Person class

Person.java

```
package com.in28minutes.oops.level2.inheritance;

public class Person {

    public Person() {
        System.out.println("Inside Person Constructor");
    }

}
```

Employee.java

```
package com.in28minutes.oops.level2.inheritance;
import java.math.BigDecimal;

public class Employee extends Person {

    public Employee() {
        //super();
        System.out.println("Inside Employee Constructor");
    }

}
```

EmployeeRunner.java

```
package com.in28minutes.oops.level2.inheritance;

public class EmployeeRunner {
    public static void main(String[] args) {
        Employee employee = new Employee();
    }
}
```

Console Output

Inside Person Constructor

Inside Employee Constructor

Snippet-02 Explained

When a sub-class object is created

- sub-class constructor is called and it implicitly invokes its super-class constructor.

The Java compiler inserts the code `super();` (if it is not explicitly added by the programmer) as the first statement in the body of the sub-class default constructor, here `Employer()` .

- The statement `super();` is the invocation of the super-class default constructor.
- Hence, the body of the super-class constructor is always invoked before the body of the sub-class constructor.

Snippet-3 : `Person` - Non-Default Constructor

Let's remove the no argument constructor and add a one argument constructor to `Person` class.

```
    public Person(String name) {
        this.name = name;
    }
```

PersonRunner.java

```
package com.in28minutes.oops.level2.inheritance;

public class PersonRunner {
    public static void main(String[] args)
        Person person = new Person("Ranga");
        person.setEmail("in28minutes@gmail.com");
        person.setPhoneNumber("123-456-7890");
        System.out.println(person);
    }
}
```

Console Output

Person Ranga , Email : in28minutes@gmail.com, Phone Number : 123-456-7890

Snippet-03 Explained

When we added the constructor with one argument for `class Employee` , the existing code in ***EmployeeRunner.java*** will cause a compilation error, because there is no longer any default constructor for `Person` !

`super()` cannot be called from within the default constructor of `Employee` .

One option is to put the no argument constructor back.

```
public Person() {
    System.out.println("Inside Person Constructor");
}
```

But, it doesn't really make sense to create a `Person` without a `name` , does it?

The solution in this case would be to call the single-argument constructor `Person(String)` by an invocation such as `super(name);`

```
    public Employee(String name, String title, String employerName, char employeeGrade) {
        super(name);
        this.title = title;
        this.employerName = employerName;
        this.employeeGrade = employeeGrade;
    }
```

EmployeeRunner.java

```java
package com.in28minutes.oops.level2.inheritance;

public class EmployeeRunner {
    public static void main(String[] args) {
        Employee employee = new Employee("Ranga", "Programmer Analyst", "In28Minutes", 'A');
        System.out.println(employee);
    }
}
```

Console Output

Employee Name: Ranga, Email: null, Phone Number: null, Title: Programmer Analyst, Employer: In28Minutes, Employee Grade: A, Salary: null

Snippet-10 : EmployeeRunner complete

Let's provide setters to set the non mandatory attributes.

EmployeeRunner.java

```java
package com.in28minutes.oops.level2.inheritance;

public class EmployeeRunner {
    public static void main(String[] args) {
        Employee employee = new Employee("Ranga", "Programmer Analyst", "In28Minutes",
        employee.setEmail("in28minutes@gmail.com");
        employee.setPhoneNumber("123-456-7890");
        employee.setSalary(new BigDecimal("50000"));
        System.out.println(employee);
    }
 }
```

Console Output

Employee Name: Ranga, Email: in28minutes@gmail.com, Phone Number: 123-456-7890, Title: Programmer Analyst, Employer: In28Minutes, Employee Grade: A, Salary: 50000.0000

Snippet-10: `Student` updated

Let's add a two argument construtor to the `Student` class.

Student.java

```java
package com.in28minutes.oops.level2.inheritance;

public class Student extends Person {
    private String collegeName;
    private int year;

    public Student(String name, String collegeName) {
        super(name);
        this.collegeName = collegeName;
    }

    public String getCollegeName() {
        return collegeName;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public int getYear() {
        return year;
    }

    public String toString() {
        return "Student : " + super.name() + ", College: " + collegeName;
    }
}
```

StudentRunner.java

```java
package com.in28minutes.oops.level2.inheritance;

public class StudentRunner {
    public static void main(String[] args)
        //Student student = new Student();
        Student student = new Student("Ranga", "IIT Bombay");
        System.out.println(student);
    }
}
```

Console Output

Student : Ranga, College : IIT Bombay

Step 13: Multiple Inheritance, Reference Variables And `instanceof`

In other programming languages, Multiple Inheritance is allowed. A class can directly inherit from two or more classes.

However, in Java, direct Multiple Inheritance is not allowed.

Snippet-01 : Multiple Inheritance

```
jshell> class Animal {
   ..>> }
| created class Animal
jshell> class Pet {
   ..>> }
| created class Pet

jshell> class Dog extends Animal, Pet {
   ..>> }
| Error:
| '{' expected
| class Dog extends Animal, Pet {
|_________________________^
jshell>**
```

`class Dog extends Animal, Pet {}` throws an error. You cannot extend two classes.

Inheritance Chains

However you can create an inheritance chain.

- `class C` **is a** `class B`
- `class B` **is a** `class A`

Let's check out a small code snippet.

Snippet-02 : An Inheritance Chain

`Dog **is a** Pet , Pet **is a** Animal```. This is an example of what is called an **inheritance hierarchy**.

```
jshell> class Animal {
   ..>> }
| created class Animal
jshell> class Pet extends Animal {
   ..>> public void groom() {
   ..>> System.out.println("Pet Groom");
   ..>> }
   ..>> }
| created class Pet_
jshell> class Dog extends Pet {
   ..>> }
| created class Dog
```

If you want, you can visualize in your mind as : *`Dog --> Pet --> Animal --> Object`* (Yes, `Object` is sitting at the top of *all* inheritance hierarchies in Java!)

The statement `Dog dog = new Dog();` sets off constructor invocations up the inheritance hierarchy: * `Dog()` invokes `Pet()` * `Pet()` invokes `Animal()` * `Animal()` invokes `Object()`

```
jshell> Dog dog = new Dog();
dog ==> Dog@23a6e47f
```

The expression `dog.toString()` does a traversal up the inheritance hierarchy as well: * Since `Dog.toString()` is not defined, the compiler looks for `Pet.toString()` * Since `Pet.toString()` is not defined, the compiler looks for `Animal.toString()` * Since `Animal.toString()` is not defined, the compiler looks for `Object.toString()` , which is always provided as the default implementation for all sub-classes of `class Object` in Java.

```
jshell> dog.toString();
$1 ==> "Dog@23a6e47f"
```

The invocation `dog.groom();` is also resolved by traversing up the inheritance hierarchy.

```
jshell> dog.groom();
"Pet Groom"
```

The statement `Pet pet = new Dog();` is really interesting. In Java, it is permitted for a *super-class reference variable to reference a sub-class object instance*.

Through such a reference, method invocations are also permitted, and the correct thing gets done. Hence, `pet.groom();` causes the output " *`Pet Groom`* ".

However, the converse assignment is not allowed. *A sub-class reference variable **cannot** reference a super-class object instance*. * The statement `Dog dog = new Pet();` therefore, causes a compiler error.

```
jshell> Pet pet = new Dog();
pet ==> Dog@22d37d54
jshell> pet.groom();
Pet Groom
```

```
jshell> Dog dog = new Pet();
| Error:
| incompatible types: Pet cannot be converted to Dog
| Dog dog = new Pet();
|___________^-------^
```

The `instanceof` operator is to find the relationship between an object and a class. If the object is an instance of the class or its sub class, it returns true.

```
jshell> pet instanceof Pet
$2 ==> true
jshell> pet instanceof Dog
$3 ==> true
```

The `instanceof` operator throws an error if the object and class are unrelated.

```
jshell> pet instanceof String
| Error:
| incompatible types: Pet cannot be converted to java.lang.String
| pet instanceof String
|_^-^
jshell> pet instanceof Animal
$4 ==> true
jshell> pet instanceof Object
$5 ==> true
```

The `instanceof` operator returns `false` if the object is an instance of a super class of the class provided.

```
jshell> Animal animal  new Animal();
animal ==> Animal@3632be31
jshell> animal instanceof Pet
$6 ==> false
jshell> animal instanceof Dog
$7 ==> false
jshell> animal instanceof Object
$8 ==> true
jshell>
```

Step 14: Introducing Abstract Classes

An `abstract class` can contain `abstract` methods.

An abstract method does not have a method definition.

Here's how a typical class looks like. You can create methods inside the class and you can create instances of the class.

```
jshell> class Animal {
   ..>> public void bark() {
   ..>> System.out.println("Animal Bark");
   ..>> }
   ..>> }
| created class Animal
jshell> Animal animal = new Animal();
animal ==> Animal@335eadca
jshell> animal.bark();
Animal Bark
```

Let's see how to create an abstract class:

```
jshell> abstract class AbstractAnimal {
   ..>> abstract public void bark();
   ..>> }
| created class AbstractAnimal
```

The syntax is simple: add `abstract` keyword before the class.

An `abstract class` cannot be instantiated.

```
jshell> AbstractAnimal animal = new AbstractAnimal();
| Error:
| AbstractAnimal is abstract; cannot be instantiated.
| AbstractAnimal animal = new AbstractAnimal();
|^----------------------------------------------^
jshell>
```

However, it can be sub-classed, creating inheritance hierarchies below it. A sub-class of an abstract class (often called a **concrete class**) must override its `abstract` methods.

```
jshell> class Dog extends AbstractAnimal {
   ..>> }
| Error:
| Dog is not abstract and does not override abstract method bark() in AbstractAnimal
| class Dog extends AbstractAnimal {
|^-----------------------------------...
jshell> class Dog extends AbstractAnimal {
   ..>> public void bark() {
   ..>> System.out.println("Bow Bow");
   ..>> }
   ..>> }
| created class Dog
jshell> Dog dog = new Dog();
dog ==> Dog@5a8e6209
jshell> dog.bark();
Bow Bow
```

Step 15: Abstract Classes - Design Aspects

Why do we need an abstract class?

Consider a feast being prepared at a home, and several dishes are on the menu for the event. Obviously, each dish would have certain procedure for it to be prepared. Cooking any dish normally involves following a tried and tested recipe, and its preparation boils down to these basic steps:

- Prepare the Ingredients
- Cook the Recipe
- Cleanup (the Mess created!)

These steps would be different for each dish but the order of steps remain the same.

Snippet-01 : The Recipe Hierarchy

Let's use abstract class to build the recipe.

AbstractRecipe.java

```
package com.in28minutes.oops.level2;
```

```java
public abstract class AbstractRecipe {
    public void execute() {
        prepareIngredients();
        cookRecipe();
        cleanup();
    }

    abstract void prepareIngredients();
    abstract void cookRecipe();
    abstract void cleanup();
}
```

We defined abstract methods for each of the steps and created an `execute` method calling them. `execute` method ensures that the order of method call is followed.

You can define implementations implementing the abstract methods.

CurryRecipe.java

```java
package com.in28minutes.oops.level2;

public class CurryRecipe extends AbstractRecipe {
    public CurryRecipe() {
        System.out.println("[Curry Preparation Method]");
    }

    @Override
    void prepareIngredients() {
        System.out.println("Get Vegetables Cut and Ready");
        System.out.println("Get Spices Ready");
    }

    @Override
    void cookRecipe() {
        System.out.println("Steam And Fry Vegetables");
        System.out.println("Cook With Spices");
        System.out.println("Add Seasoning");
    }

    @Override
    void cleanup() {
        System.out.println("Discard unused Vegetables");
        System.out.println("Discard unused Spices");
    }
}
```

RecipeRunner.java

```java
package com.in28minutes.oops.level2;

public class RecipeRunner {
    public static void main(String[] args) {
        CurryRecipe curryRecipe = new CurryRecipe();
        curryRecipe.execute();
    }
}
```

Console Output

[Curry Preparation Method]

Get Vegetables Cut and Ready

Get Spices Ready

Steam And Fry Vegetables

Cook With Spices

Add Seasoning

Discard unused Vegetables

Discard unused Spices

Snippet-01 Explained

`CurryRecipe` defines what needs to be done in each step. When we invoke the `execute` method, the steps are executed in order.

Snippet-02 : MicrowaveCurryRecipe

We can easily create more recipes.

MicrowaveCurryRecipe.java

```java
package com.in28minutes.oops.level2;

public class MicrowaveCurryRecipe extends AbstractRecipe {
    public MicrowaveCurryRecipe() {
        System.out.println("[Curry Microwave Method]");
    }

    @Override
    void prepareIngredients() {
        System.out.println("Get Vegetables Cut and Ready");
        System.out.println("Switch on Microwave");
    }

    @Override
    void cookRecipe() {
        System.out.println("Microwave Vegetables");
        System.out.println("Add Seasoning");
    }

    @Override
    void cleanup() {
        System.out.println("Switch Off Microwave");
        System.out.println("Discard unused Vegetables");
    }
}
```

RecipeRunner.java

```java
package com.in28minutes.oops.level2;

public class RecipeRunner {
    public static void main(String[] args) {
        MicrowaveCurryRecipe mcirowaveRecipe = new MicrowaveCurryRecipe();
        microwaveRecipe.execute();
    }
}
```

Console Output

[Curry Microwave Method]

Get Vegetables Cut and Ready

Switch on Microwave

Microwave Vegetables

Add Seasoning

Switch off Microwave

Discard unused Vegetables

Snippet-02 Explained

`MicrowaveCurryRecipe` defines what needs to be done in each step. When we invoke the `execute` method, the steps are executed in order.

Summary

This pattern is called a `Template method` pattern. You define an abstract class with the order of steps defined. You leave the implementation of each of the steps to the sub classes.

Step 16: Abstract Classes - Puzzles

Let's look a few FAQ regarding abstract classes.

An `abstract class` can be created, without any `abstract` member methods.

```
jshell> abstract class AbstractTest {
   ...>}
| creates abstract class AbstractTest
```

An `abstract class` can be a sub class to create another `abstract class` , without overriding any of the super-class `abstract` methods.

```
jshell> abstract class AbstractAlgorithm {
   ...> abstract void flowChart();
   ...> }
| creates abstract class AbstractAlgorithm
jshell> abstract class AlgorithmTypeOne extends AbstractAlgorithm {
   ...> }
| creates abstract class AlgorithmTypeOne
```

An `abstract class` can have member variables.

```
jshell> abstract class AbstractAlgorithm {
   ...> private int stepCount;
   ...> }
| replaced abstract class AbstractAlgorithm
```

An `abstract class` can have non-abstract methods.

```
jshell> abstract class AbstractAlgorithm {
   ...> private int stepCount;
   ...> public int getStepCount() {
```

```
   ...> return stepCount;
   ...> }
   ...> }
|  replaced abstract class AbstractAlgorithm
jshell>
```

Step 17: Introducing Interfaces

What does a "*gaming console*" mean to you?

A device that has buttons or other controls on it, that enable us to play video games.

What are the typical buttons on it?

- Arrows - up down left right
- Select
- etc

Gaming console offers an interface to play your games.

Who provides the implementation of what happens when a button is clicked? The game implementor - for example, the implementor of Mario game or the Chess game.

How do you represent this in Java program?

GamingConsole.java

```
package com.in28minutes.oops.level2.interfaces;

public interface GamingConsole {
    public void up();
    public void down();
    public void left();
    public void right();
}
```

`interface GamingConsole` contains methods without definitions.

Who provides the implementations?

Welcome `MarioGame`.

MarioGame.java

```
package com.in28minutes.oops.level2.interfaces;

public class MarioGame implements GamingConsole {
    @Override
    public void up() {
        System.out.println("Jump");
    }

    @Override
    public void down() {
        System.out.println("Go into a hole");
    }

    @Override
    public void left() {
    }

    @Override
    public void right() {
        System.out.println("Go Forward");
    }
}
```

`MarioGame` provides a definition for all the methods declared in the interface `GamingConsole`. Syntax is simple. `class MarioGame implements GamingConsole` and implement all the methods.

Let's look at how you can run these games.

GameRunner.java

```
package com.in28minutes.oops.level2.interfaces;

public class GameRunner {
    public static void main(String[] args) {
        MarioGame game = new MarioGame();
        game.up();
        game.down();
        game.left();
        game.right();
    }
}
```

Console Output

Jump

Go into a hole

Go forward

Snippet-01 Explained

The main advantage of having an interface is that it can be used to enforce a contract for its implementors.

Snippet-02 : Code Reuse With Interfaces

Let's look at another example - `ChessGame`.

ChessGame.java

```
package com.in28minutes.oops.level2.interfaces;

public class ChessGame implements GamingConsole {

    @Override
    public void up() {
        System.out.println("Move Piece Up");
    }

    @Override
    public void down() {
        System.out.println("Move Piece Down");
    }

    @Override
    public void left() {
        System.out.println("Move Piece Left");
    }

    @Override
    public void right() {
        System.out.println("Move Piece Right");
    }
}
```

Running it is simple. All that you need to do is to comment out `MarioGame game = new MarioGame();` and replace it with an implementation of `ChessGame` .

GameRunner.java

```
package com.in28minutes.oops.level2.interfaces;

public class GameRunner {
    public static void main(String[] args) {
        //MarioGame game = new MarioGame();
        ChessGame game = new ChessGame();
        game.up();
        game.down();
        game.left();
        game.right();
    }
}
```

Console Output

Move Piece Up

Move Piece Down

Move Piece Left

Move Piece Right

Snippet-2 explained

In the same `GamerRunner` `class` , if we now instantiate a `ChessGame` object instead of a `MarioGame` one, none of the other code needs to change. This is because both `MarioGame` and `ChessGame` implement the same `interface` , `GamingConsole` .

Using Interface as type for reference variable

`GamingConsole` is an interface. `MarioGame` and `ChessGame` are it's implementations.

Let's try this - `GamingConsole game = new ChessGame();`

GameRunner.java

```
package com.in28minutes.oops.level2.interfaces;

public class GameRunner {
    public static void main(String[] args) {
        GamingConsole game = new ChessGame();
        game.up();
        game.down();
        game.left();
        game.right();
    }
}
```

Console Output

Move Piece Up

Move Piece Down

Move Piece Left

Move Piece Right

Explained

`GamingConsole game = new ChessGame();` - You can store an implementation of an interface into a reference variable of the type of the interface.

How does this help?

Let's look at the next example:

Snippet-4 : GameRunner Version 4

GameRunner.java

```java
package com.in28minutes.oops.level2.interfaces;

public class GameRunner {
    public static void main(String[] args) {
        GamingConsole game = new MarioGame();
        game.up();
        game.down();
        game.left();
        game.right();
    }
}
```

Console Output

Jump

Go into a hole

Go forward

Snippet-04 Explained

You can replace `GamingConsole game = new ChessGame()` with `GamingConsole game = new MarioGame()` and now the program runs the `MarioGame`. Isn't it awesome?

Step 18: Using Interfaces To Design APIs

Consider a Software Development project, which involves programming a fairly large and complex application. Project team (Team A) decided to out-source part of this project to an external team (Team B). Let's say this external team needs to implement a farily complex algorithm to achieve a specific task, and which needs to interface with the rest of the application. Work on both parts of the application needs to proceed simultaneously.

Suppose the algorithm logic is implemented using a single method:

```java
int complexAlgorithm(int number1, int number2);
```

How do we ensure that the work done by both the teams remains compatible?

They start with defining an interface.

ComplexAlgorithm.java

```
package com.in28minutes.oops.level2.interfaces;

public interface ComplexAlgorithm {
    int complexAlgorithm(int number1, int number2);
}
```

Now the teams can go on their merry way. Team A can create a stub for the interface `OneComplexAlgorithm` and start working on their project.

OneComplexAlgorithm.java

```
package com.in28minutes.oops.level2.interfaces;

public class OneComplexAlgorithm {
    public int complexAlgorithm(int number1, int number2) {
        return number1 + number2;
    }
}
```

Team B can take time to implement the actual algorithm.

ActualComplexAlgorithm.java

```
package com.in28minutes.oops.level2.interfaces;

public class ActualComplexAlgorithm {
    public int complexAlgorithm(int number1, int number2) {
        //Your complex implementation will be present here..
        return number1 * number2;
    }
}
```

Step 19: Interfaces - Puzzles And Interesting Facts

Let's look at a few examples to understand interfaces further.

An `interface` can be extended by another `interface`. We can have an inheritance hierarchy purely consisting of interfaces.

```
jshell> interface InterfaceOne {
   ...> void methodOne();
   ...> }
| created interface InterfaceOne

jshell> interface InterfaceTwo extends InterfaceOne {
   ...> void methodTwo();
   ...> }
| created interface InterfaceTwo
```

An implementation of interface should implement all its methods including the methods in its super interfaces.

```
jshell> class Implementation implements InterfaceTwo {
   ...>}
| Error:
| Implementation is not abstract and does not implement abstract method methodTwo() of     Inter
| class Implementation implements InterfaceTwo {
|^---------------------------------------------...

jshell> public class Implementation implements InterfaceTwo {
   ...> public void methodTwo() {}
   ...> }
| Error:
| Implementation is not abstract and does not implement abstract method methodOne() of     Inter
| class Implementation implements InterfaceTwo {
|^---------------------------------------------...

jshell> public class Implementation implements InterfaceTwo {
   ...> public void methodTwo() {}
   ...> public void methodOne() {}
   ...> }
| created class Implementation
```

If a class is declared as abstract, it can skip providing implementations for all interface methods.

```
jshell> public abstract class AbstractImplementation implements InterfaceTwo {
   ...> public void methodOne() {}
   ...> }
| created class AbstractImplementation
```

`interfaces` cannot have member variables. An `interface` can only have declared **constants**

```
jshell> interface InterfaceThree {
   ...> int test;
   ...> }
| Error:
| = expected
| int test;
|_________^

jshell> interface InterfaceThree {
   ...> int test = 5;
   ...> }
| created interface InterfaceThree
```

Starting from Java SE 8, an interface can provide a default implementation of the methods it provides. It can be done by including the keyword `default` in the signature of that method, and providing a body to that method in its definition.

```
jshell> interface InterfaceFour {
   ...> public default void print() {
   ...> System.out.println("default print");
   ...> }
   ...> }
| created interface InterfaceFour

jshell> class TestPrint implements InterfaceFour {
   ...> }
| created class TestPrint


jshell> TestPrint testPrint = new TestPrint();
testPrint ==> TestPrint@6ebc05a6
```

```
jshell> testPrint.print();
default print
```

Implementations of `interface` can override the default method implementation.

```
jshell> class ParticularPrint implements InterfaceFour {
   ...> public void print() {
   ...> System.out.println("particular print");
   ...> }
   ...> }
| created class ParticularPrint
```

```
jshell> ParticularPrint particularPrint = new ParticularPrint();
particularPrint ==> ParticularPrint@5fad14c4
jshell> particularPrint.print();
particular print
jshell>
```

No method declared inside an `interface` can be qualified with the `private` access specifier. However, an `abstract class` can have `private` methods declared within.

Why do we need `default` method implementations.

Let's consider a `Provider` interface with three implementations.

```
public interface Provider {
    public void doSomething();
}

public class ImplementorOne {
    @Override
    public void doSomething() {
        System.out.println("Do One");
    }
}

public class ImplementorTwo {
    @Override
    public void doSomething() {
        System.out.println("Do Two");
    }
}

public class ImplementorThree {
    @Override`
    public void doSomething() {
        System.out.println("Do Three");
    }
}
```

What happens if a new method is added to the `interface` ?

```
public interface Provider {
    public void doSomething();
    public void doMore();
}
```

Compilation Error! All implementations classes `ImplementationOne` , `ImplementationTwo` and `ImplementationThree` **must** me updated to implement `doMore()` in each case.

Alternative : provide a default implementation for `doMore` .

```
public interface Provider {
    public void doSomething();

    public default void doMore(){
        System.out.println("Do More");
    }
}
```

No other code needs to immediately change, and specific implementation classes can override this default version, as and when needed.

This is especially useful in building and extending frameworks. You are not breaking a user of your framework interface when you add new methods to the interface.

Step 20: `abstract class` And `interface` : A Comparison

`abstract class` and `interface` are very different, except that they have a very similar syntax.

When would you want to use them in your application?

interface

`interface` is a **Contract**.

An `interface` is primarily used when you have two software components that need to communicate with each other, and there is a need to establish a contract.

Recall the following example: `ComplexAlgorithm` defines the interface which helps both the teams involved.

```
package com.in28minutes.oops.level2.interfaces;
public interface ComplexAlgorithm {
    int complexAlgorithm(int number1, int number2);
}
```

abstract class

An `abstract class` is primarily used when you want to generalize behavior by creating a super class.

Recall the following example we had discussed:

```
public abstract class AbstractRecipe {
    public void execute() {
        prepareIngredients();
        cookRecipe();
        cleanup();
    }

    abstract void prepareIngredients();
    abstract void cookRecipe();
    abstract void cleanup();
}
```

Syntactical Comparison

Here are important syntactical differences:

- No method declared inside an `interface` can be qualified with a `private` access specifier. However, an `abstract class` can have `private` methods.
- An `interface` cannot have declared member variables. An `abstract class` can have member variable declarations.
- A `class` or an `abstract class` can implement multiple `interface` s. But, an `interface` can extend only one `interface` , and a `class` or an `abstract class` can extend only one `class` or `abstract class` .

Step 21: Programming Exercise PE-OOP-03

Exercises

TODO

Solution To PE-OOP-03

Solution - 1

Solution - 2

Step 21: Introducing Polymorphism

TODO

Introducing Collections

Arrays are not dynamic data structures. They can store only a fixed maximum number of elements.

Inserting or removing elements into an array needs a lot of work.

Collections are in-built implementations available to support dynamic data structures in Java programs. The commonly used collections are based on Lists, Trees and Hash Tables.

Java provides very good collection implementations.

To make it simple to understand all collections, we will start with the interfaces - such as `List` , `Set` , `Queue` , `Map` and others.

After that, we will look at implementations of these interfaces, such as `LinkedList` , `HashTable` and `TreeMap` , among others.

The Collection Interfaces

Let's start with the `List` interface

The `List interface`

The `List interface` is used to implement an **ordered collection** in Java programs. An ordered collection is one, where the programmer has control over the position where an element can be inserted into, or accessed from the collection.

If an element's insertion position is not specified, it is added at the end of the `List` .

A `List` typically allows *duplicate* elements.

Snippet-1 : Creating a List

We can see examples of creating a list and accessing element data below:

```
jshell> List<String> words = List.of("Apple", "Bat", "Cat");
words ==> [Apple, Bat, Cat]
jshell> words.size()
$1 ==> 3
jshell> words.isEmpty()
$2 ==> false
jshell> words.get(0)
$3 ==> Apple
jshell> words.contains("Dog")
$4 ==> false
jshell> words.contains("Cat")
$5 ==> true
jshell> words
words ==> [Apple, Bat, Cat]
jshell> words.indexOf("Cat")
$6 ==> 2
jshell> words.indexOf("Dog")
$7 ==> -1
jshell>
```

`List` Immutability

Consider the `List` `words` we created in the last snippet.

```
List<String> words = List.of("Apple", "Bat", "Cat");
```

Lists created using the `static` `of` method are *immutable*.

```
jshell> List<String> words = List.of("Apple", "Bat", "Cat");
words ==> [Apple, Bat, Cat]
jshell> words.add("Dog");
| java.lang.UnsupportedOperationExcetion thrown:
| at ImmutableCollections.uoe (ImmutableCollections.java:71)
| at ImmutableCollections$AbstractImmutableList.add (ImmutableCollections.java:77)
| at (#15:1)
jshell>
```

Creating A Mutable `List` s

The way to create `List` data structures that can be updated over time, is to instantiate built-in `collection` classes that implement the `List` interface.

Examples are `ArrayList` , `LinkedList` and `Vector` .

Snippet-3: Mutable Lists

Let's look at a few examples:

```
jshell> List<String> words = List.of("Apple", "Bat", "Cat");
words ==> [Apple, Bat, Cat]

jshell> List<String> wordsArrayList = new ArrayList<String>(words);
wordsArrayList ==> [Apple, Bat, Cat]

jshell> List<String> wordsLinkedList = new LinkedList<String>(words);
wordsLinkedList ==> [Apple, Bat, Cat]

jshell> List<String> wordsVector = new Vector<String>(words);
```

```
wordsVector ==> [Apple, Bat, Cat]

jshell> wordsArrayList.add("Dog");
$1 ==> true

jshell> wordsArrayList
wordsArrayList ==> [Apple, Bat, Cat, Dog]

jshell>
```

`ArrayList` vs `LinkedList`

`ArrayList` uses an array to store elements.

- Positional access and modification of elements is very efficient, with constant-time algorithmic complexity.
- Insertion and deletion of elements are expensive. In a 20 element list, to insert an element at first position, all 20 elements should be moved.

The data structure used to implement a `LinkedList` is of the type linked-list, which is a chain of blocks of memory slots.

- Inserting and Deleting values is easy. This is because in a chain of blocks, each link is nothing but a reference to the next block. Insertion only involves adjustment of these links to accommodate new values, and does not require extensive copying and shifting of existing elements.
- Positional access and modification of elements is less efficient than in an `ArrayList` , because access always involves traversal of links.

Optimization: the underlying data structure for a `LinkedList` is actually a doubly-linked list, with each element having both forward and backward links to elements around it.

`Vector` vs `ArrayList`

`Vector` has been with Java since v1.0, whereas `ArrayList` was added later, in v1.2. Both of them use an array as the underlying data structure.

`Vector` is thread-safe. In `Vector` , all methods are `synchronized` .

> Look at other thread safe `List` implementations as `Vector` has poor concurrency.

`List` Operations

We will examine common `List` operations on an `ArrayList` . Similar operations can also be done on a `LinkedList` or a `Vector` .

Snippet-4 : List Insertions

We look at examples for

- Insertion at end (default)
- Positional insertion
- Inserting duplicate entries is allowed
- Adding all elements of an external list, to the current list
  - At the end
  - Positional insertion is also available, as in single element insertion

```
jshell> List<String> words = List.of("Apple", "Bat", "Cat");
words ==> [Apple, Bat, Cat]
jshell> List<String> wordsArrayList = new ArrayList<String>(words);
wordsArrayList ==> [Apple, Bat, Cat]
```

```
jshell> wordsArrayList.add("Dog");
$1 ==> true
jshell> wordsArrayList
wordsArrayList ==> [Apple, Bat, Cat, **Dog**]
jshell> wordsArrayList.add("Elephant");
$2 ==> true
jshell> wordsArrayList
wordsArrayList ==> [Apple, Bat, Cat, Dog, **Elephant**]
jshell> wordsArrayList.add(2, "Ball");
*jshell> wordsArrayList
wordsArrayList ==> [Apple, Bat, **Ball**, Cat, Dog, Elephant]
jshell> wordsArrayList.add("Ball");
jshell> wordsArrayList
wordsArrayList ==> [Apple, Bat, Ball, Cat, Dog, Elephant, **Ball**]
jshell> List<String> newList = List.of("Yak", "Zebra");
newList ==> [Yak, Zebra]
jshell> wordsArrayList.addAll(newList);
$3 ==> true
jshell> wordsArrayList
wordsArrayList ==> [Apple, Bat, Ball, Cat, Dog, Elephant, Ball,  **Yak**, **Zebra**]
jshell>
```

Snippet-5 : Element Modification

Let's look at examples of

- Modifying elements at a specific position.
- Deleting elements

```
jshell> wordsArrayList
wordsArrayList ==> [Apple, Bat, Ball, Cat, Dog, Elephant, Ball,  Yak, Zebra]
jshell> wordsArrayList.set(6, "Fish");
$4 ==> "Ball"
jshell> wordsArrayList
wordsArrayList ==> [Apple, Bat, Ball, Cat, Dog, Elephant, **Fish**,  Yak, Zebra]
jshell> wordsArrayList.remove(2);
$5 ==> "**Ball**"
jshell> wordsArrayList
wordsArrayList ==> [Apple, Bat, Cat, Dog, Elephant, Fish,  Yak, Zebra]
jshell> wordsArrayList.remove("Dog");
$6 ==> true
jshell> wordsArrayList
wordsArrayList ==> [Apple, Bat, Cat, Elephant, Fish,  Yak, Zebra]
jshell> wordsArrayList.remove("Dog");
$6 ==> false
jshell>
```

Snippet-6 : Iterating within ArrayList

Let's now look at how to iterate around the contents of a `List`

Basic for loop:

```
jshell> List<String> words = List.of("Apple", "Bat", "Cat");
words ==> [Apple, Bat, Cat]
jshell> for(int i=0; i<words.size(); i++) {
...> System.out.println(words.get(i));
...> }
Apple
Bat
Cat
```

Using enhanced for loop:

```
jshell> for(String word:words) {
...> System.out.println(word);
...> }
Apple
Bat
Cat
```

Using iterators:

```
jshell> Iterator wordsIterator = words.iterator();
wordsIterator ==> java.util.AbstractList$Itr@3712b94
jshell> while(wordsIterator.hasNext()) {
...> System.out.println(wordsIterator.next());
...> }
Apple
Bat
Cat
```

Choosing Between Iteration Modes

- For non-mutating iterations, an enhanced `for` loop is the most convenient
- For mutating iterations:
- Deletions: An enhanced `for` loop is not recommended, as we saw in the example. `"Bat"` was removed, but `"Cat"` was not, because the iteration itself was affected.
- Iterators can be used for such iterations.

Snippet-6

```
jshell> List<String> words = List.of("Apple", "Bat", "Cat");
words ==> [Apple, Bat, Cat]
jshell> List<String> wordsAL = new ArrayList<>(words);
wordsAL ==> [Apple, Bat, Cat]
jshell> for(String word:words) {
  ...> if(word.endsWith("at"))
   ...> System.out.println(word);
   ...> }
   ...> }
Bat
Cat
jshell> for(String word:wordsAL) {
   ...> if(word.endsWith("at"))
   ...> wordsAL.remove(word);
   ...> }
   ...> }
jshell> wordsAL
wordsAL ==> [Apple, Cat]
jshell> Iterator wordsIterator = wordsAL.iterator();
wordsIterator ==> java.util.AbstractList$Itr@3712b94
jshell> while(wordsIterator.hasNext()) {
   ...> if(wordsIterator.next().endsWith("at")){
   ...> wordsIterator.remove();
   ...> }
   ...> }
jshell> wordsAL
wordsAL ==> [Apple]
jshell>
```

`List` : Type Safety

A `List` collection does not store primitives. It only stores object references. When we attempt to store primitive types, such as those of `int` , `char` or `double` , they get implicitly converted to their wrapper class types, namely `Integer` , `Character` and `Double` respectively. Primitive data type elements get *auto-boxed*.

Snippet-7 : Type safety

The `ArrayList.indexOf()` method is not overloaded, there is only one version. SO, when passed an `int` argument, it is auto-boxed to an `Integer` object, and the method gets executed.

- However, the `ArrayList.remove()` method has two overloaded versions:
  - `ArrayList.remove(int index)` : attempts to delete an element at the specified index in the `ArrayList`
  - `ArrayList.remove(Object o)` : attempts to delete the specified element, if ti is present in the `ArrayList` .

```
jshell> List values = List.of("A", 'A', 1, 1.0);
values ==> ["A", 'A', 1, 1.0]
jshell> values.get(2)
$1 ==> 1
jshell> values.get(2) instanceof Integer
$2 ==> true
jshell> values.get(1) instanceof Character
$3 ==> true
jshell> values.get(3) instanceof Double
$4 ==> true
jshell> List<String> textValues = List.of("A", 'A', 1, 1.0);
| Error
| Error
| List<String> textValues = List.of("A", 'A', 1, 1.0);
|___________________________^________________________^
jshell> List<Integer> numbers = List.of(101, 102, 103, 104, 105);
numbers ==> [101, 102, 103, 104, 105]
jshell> numbers.indexOf(101)
$5 ==> 0
jshell> List<Integer> numbersAL = new ArrayList<>(numbers);
numbersAL ==> [101, 102, 103, 104, 105]
jshell> numbersAL.indexOf(101)
$6 ==> 0
jshell> numbersAL.remove(101)
java.lang.IndexOutOfBoundsException!!
jshell> numbersAL.remove(Integer.valueOf(101))
$7 ==> true
jshell> numbersAL
numbersAL ==> [102, 103, 104, 105]
jshell>
```

Sorting a `List`

Snippet-8 : List sort

The `List` obtained from `List.of()` is immutable, hence cannot be sorted itself. Hence, create a mutable `ArrayList` out of it.

The `ArrayList.sort()` method requires the definition of a `Comparator` object. Easier option is to use `Collections.sort()` instead.

```
jshell> List<Integer> numbers = List.of(123, 12, 3, 45);
numbers ==> [123, 12, 3, 45]
jshell> List<Integer> numbersAL = new ArrayList<>(numbers);
numbersAL ==> [123, 12, 3, 45]
```

```
jshell> numbersAL.sort();
| Error:
| required: java.util.Comparator<? super java.lang.Integer>
| numbersAL.sort();
|^--------------^
jshell> Collections.sort(numbersAL);
jshell> numbersAL
numbersAL ==> [3, 12, 45, 123]
jshell>
```

Sorting List

Snippet-9 : Sorting List

Let's create a Student class and try to sort it using `Collections.sort`

Student.java

```java
package collections;

public class Student {
    private int id;
    private String name;

    public Student(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String toString() {
        return id + " " + name;
    }
}
```

StudentsCollectionRunner.java

```java
package collections;
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;

public class StudentsCollectionRunner {
    public static void main(String[] args) {
        List<Student> students = List.of(new Student(1, "Ranga"),
                                         new Student(100, "Adam"),
                                         new Student(2, "Eve"));
```

```
            ArrayList<Student> studentsAl = new ArrayList<>(students);
            System.out.println(studentsAl);
            Collections.sort(studentsAl);
            System.out.println(studentsAl);
        }
    }
```

Console Output

COMPILER ERROR

Snippet-9 Explained

To the method `Collections.sort()`, only those `List`s can be passed, the type of whose elements `T`, implements the `Comparator<? super T> interface`.

`Student` does not implement the interface. Result - Compilatino error.

Snippet-10 : Implementing Comparator Interface

StudentsCollection.java

```
    package collections;
    import java.util.Comparable;

    public class Student implements Comparable<Student> {
        // Same as earlier
        @Override
        public int compareTo(Student that) {
            return Integer.compare(this.id, that.id);
        }
    }
```

StudentsCollectionRunner.java

```
    package collections;
    import java.util.Collections;
    import java.util.List;
    import java.util.ArrayList;

    public class StudentsCollectionRunner {
        public static void main(String[] args) {
            List<Student> students = List.of(new Student(1, "Ranga"),
                                            new Student(100, "Adam"),
                                            new Student(2, "Eve"));
            ArrayList<Student> studentsAl = new ArrayList<>(students);
            System.out.println(studentsAl);
            Collections.sort(studentsAl);
            System.out.println(studentsAl);
        }
    }
```

Console Output

[1 Ranga, 100 Adam, 2 Eve]

[1 Ranga, 2 Eve, 100 Adam]

Snippet-10 Explained

`Integer.compare(x, y)` returns the value 0 if x == y; a value less than 0 if x < y; and a value greater than 0 if x > y.

If you change the order of parameters:

```
@Override
public int compareTo(Student that) {
    return Integer.compare(that.id, this.id);
}
```

Output changes to:

[100 Adam, 2 Eve, 1 Ranga]

The `Comparator` interface

What if there is a need for sorting Student's in multiple ways? What if we want to sort Students in ascending order of id's in some situations and in descending order of id's in some other situations?

`Collections.sort` has an overloaded version that has a signature that looks like:

```
@SuppressWarnings({"unchecked", "rawtypes" })
public static <T> sort(List<T> list, Comparator<? super T> c)
    list.sort(c);
}
```

To be able to invoke this version of `Collections.sort` , we will need to define a suitable `Comparator` implementation, that works with `Student` objects.

As you may have already guessed, we would need to define two different `Comparator` implementations: one for ascending sort, and another for descending sort.

Snippet-11 : Implementing Student Comparators

```
package collections;
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
import java.util.Comparator;

class DescStudentComparator implements Comparator<Student> {
    @Override
    public int compare(Student student1, Student student2) {
        return Integer.compare(student2.getId(), student1.getId());
    }
}



public class StudentsCollectionRunner {
    public static void main(String[] args) {
        List<Student> students = List.of(new Student(1, "Ranga"),
                                        new Student(100, "Adam"),
                                        new Student(2, "Eve"));
        ArrayList<Student> studentsAl = new ArrayList<>(students);
        System.out.println(studentsAl);
        Collections.sort(studentsAl);
        System.out.println("Ascending : " + studentsAl);
        Collections.sort(studentsAl, new DescStudentComparator());
        System.out.println("Descending : " studentsAl);
    }
}
```

Console Output

[1 Ranga, 100 Adam, 2 Eve]

Ascending : [1 Ranga, 2 Eve, 100 Adam]

Descending : [100 Adam, 2 Eve, 1 Ranga]

`sort` method Of `List`

You can call `sort` method on the `List` by passing it a `Comparator` implementation - `studentsAl.sort(new AscStudentComparator())`.

Snippet-12 : Comparator for ArrayList.sort()

StudentsCollectionRunner.java

```
package collections;
import java.util.Collections;
import java.util.List;
import java.util.ArrayList;
import java.util.Comparator;

public class StudentsCollectionRunner {
    public static void main(String[] args) {
        List<Student> students = List.of(new Student(1, "Ranga"),
                                        new Student(100, "Adam"),
                                        new Student(2, "Eve"));

        ArrayList<Student> studentsAl = new ArrayList<>(students);
        System.out.println(studentsAl);
        studentsAl.sort(new AscStudentComparator());
        System.out.println("Asc : " + studentsAl);
        studentsAl.sort(new DescStudentComparator());
        System.out.println("Desc : " + studentsAl);
    }
}
```

Console Output

[1 Ranga, 100 Adam, 2 Eve]

Asc : [1 Ranga, 2 Eve, 100 Adam]

Desc : [100 Adam, 2 Eve, 1 Ranga]

The `Set` `interface`

Mathematically, a set is a collection of unique items. Similarly, the Java `Set` `interface` also specifies a contract for collections having unique elements.

- If `object1.equals(object2)` returns `true`, then only one of `object1` and `object2` have a place in a `Set` implementation.

There is no positional element access provided by the `Set` implementations.

Snippet-13 : Set

Let's look at a few examples:

The collection returned by `Set.of()` is immutable, hence does not support the `add()` method.

```
jshell> Set<String> set = Set.of("Apple", "Banana", Cat");
set ==> [Banana, Apple, Cat]
```

Create a `HashSet` collection instead, which supports the `add()`, in order to test the uniqueness property of a `Set`.

```
jshell> set.add("Apple");
| java.lang.UnsupportedOperationException thrown:
jshell> Set<String> hashSet = new HashSet<>(set);
hashSet ==> [Apple, Cat, Banana]
```

The `HashSet.add()` operation returns a `false` value, indicating that inserting a duplicate "Apple" entry has failed.

```
jshell> hashSet.add("Apple");
$1 ==> false
jshell> hashSet
hashSet ==> [Apple, Cat, Banana]
```

Note that when the `hashSet` was constructed using the `set`, the order of the elements got permuted/changed. Also originally, when we created the `set` from the `Set.of()` method, the order printed was different from the order of initialization. This confirms the fact that a `Set` collection does not give any importance to element order, and therefore, does not support positional access. Hence, the compiler error on call to `hashSet.add(int, String)`.

```
jshell> hashSet.add(2, "Apple");
| Error:
| no suitable method found for add(int, java.lang.String)
| hashSet.add(2, "Apple");
|^----------^
jshell>
```

Understanding Data Structures

Hash Table

- Hash Table: A data structure that attempts to combine the best of both worlds:
  - Very efficient element access
  - Very efficient element insertion and deletion
- Buckets : They are the slots in the hash table, into which elements are inserted and chained together. As you can see, a hash table is effectively a large array of buckets, each containing a small linked list.
- Hashing: A procedure called **hashing** used in the construction of the Hash Table. The term **hash** generally means *mix up the order of elements*.
- Hashing Function: A formula to determine/compute which bucket a particular element gets inserted into. For illustration: `hash(elem)` could compute `elem mod 13` mathematically, and uses that value to index into the table, to locate the bucket insertion. The `Object.hashCode()` could be used as a hashing function.
- Collisions: Leads to chaining within a bucket, where a linked list grows. The larger the table, and the cleverer the hashing function, the lesser the chance for collisions.

Tree

- Tree: Stores elements in sorted order. For every node in the tree, elements of all nodes in its left sub-tree are lesser than its contained element. Conversely, elements of all nodes in its right sub-tree are greater than its contained element.

- Insertions: Given any node, we know by comparing the new element with the node's element, where to go about inserting it, to maintain sorted order.
- Access : More efficient than linked lists.

`Set` Implementations

- HashSet
- LinkedHashSet
- TreeSet

Snippet-14 : HashSet

In a `HashSet` , elements are neither stored in the order of insertion, nor in sorted order.

```
jshell> Set<Integer> numbers = new HashSet<>();
numbers ==> []
jshell> numbers.add(765432);
$1 ==> true
jshell> numbers.add(76543);
$2 ==> true
jshell> numbers.add(7654);
$3 ==> true
jshell> numbers.add(765);
$4 ==> true
jshell> numbers.add(76);
$5 ==> true
jshell> numbers
numbers ==> [765432, 7654, 76, 765, 76543]
jshell>
```

Snippet-15 : LinkedHashSet

In a `LinkedHashSet` , elements are stored in the order of insertion.

```
jshell> Set<Integer> numbers = new LinkedHashSet<>();
numbers ==> []
jshell> numbers.add(765432);
$1 ==> true
jshell> numbers.add(76543);
$2 ==> true
jshell> numbers.add(7654);
$3 ==> true
jshell> numbers.add(765);
$4 ==> true
jshell> numbers.add(76);
$5 ==> true
jshell> numbers
numbers ==> [765432, 76543, 7654, 765, 76]
jshell> numbers.add(7654321);
$5 ==> true
jshell> numbers
numbers ==> [765432, 76543, 7654, 765, **7654321**]
jshell>
```

Snippet-16 : TreeSet

In a `TreeSet` , elements are stored in sorted order.

```
jshell> Set<Integer> numbers = new TreeSet<>();
numbers ==> []
jshell> numbers.add(765432);
$1 ==> true
jshell> numbers.add(76543);
$2 ==> true
jshell> numbers.add(7654);
$3 ==> true
jshell> numbers.add(765);
$4 ==> true
jshell> numbers.add(76);
$5 ==> true
jshell> numbers
numbers ==> [76, 765, 7654, 76543, 765432]
jshell> numbers.add(7);
$5 ==> true
jshell> numbers
numbers ==> [**7**, 76, 765, 7654, 76543, 765432]
jshell>
```

Exercise Set

1. Create a ```List`` of characters, such as:

```
List<Character> list = List.of('A', 'Z', 'A', 'B', 'Z, 'F);
```

- Write a procedure to list out the unique characters in this list
- Write a procedure to list out these unique characters in sorted order
- Write a procedure to list out these unique characters in the order in which they were present in the original list

Solution

SetRunner.java

```
package collections;
import java.util.List;
import java .util.Set;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.TreeSet;

public class SetRunner {
    public static void main(String[] args) {
        List<Character> characters = List.of('A', 'Z', 'A', 'B', 'Z, 'F);
        Set<Character> hashSetChars = new HashSet<>(characters);
        System.out.println("Unique Characters: " + hashSetChars);
        Set<Character> treeSetChars = new TreeSet<>(characters);
        System.out.println("Sorted Order: " + treeSetChars);
        Set<Character> linkedSetChars = new LinkedHashSet<>(characters);
        System.out.println("Inserted Order: " + linkedSetChars);
    }
}
```

Console Output

Unique Characters: [A, B, F, Z]

Sorted Order: [A, B, F, Z]

Inserted Order: [A, Z, B, F]

Solution Explained

In this example, the order to elements traversed in `hashSetChars` happened to be the same as that in `treeSetChars`. Such an order is not always guaranteed, as we saw in the earlier examples.

`TreeSet` In Depth

- Super-Interfaces
  - Set
  - NavigableSet

Snippet-16 : NavigableSet Operations

Let's look at few operations:

```
    jshell> TreeSet<Integer> numbers = new TreeSet<>(Set.of(65, 54, 34, 12, 99));
    numbers ==> [12, 34, 54, 65, 99]
    jshell> numbers.floor(40);
    $1 ==> 34
```

`floor()` method is inclusive, `lower()` is exclusive

```
    jshell> numbers.floor(34);
    $2 ==> 34
    jshell> numbers.lower(34);
    $3 ==> 12
```

`ceiling()` method is inclusive, `higher()` is exclusive

```
    jshell> numbers.ceiling(36);
    $4 ==> 54
    jshell> numbers.ceiling(34);
    $5 ==> 34
    jshell> numbers.higher(34);
    $6 ==> 54
```

`subSet(Object, Object)` method is only lower-inclusive. So, the left bound is like `lower()`, and right bound is like `higher()`. The overloaded version `subSet(Object, boolean, Object, boolean)` can be used to configure lower- and upper inclusiveness of the returned subset.

```
    jshell> numbers.subSet(20, 80);
    $7 ==> [34, 54, 65]
    jshell> numbers.subSet(34, 54);
    $8 ==> [34]
    jshell> numbers.subSet(34, 65);
    $9 ==> [34, 54]
    jshell> numbers.subSet(34, true, 65, true);
    $10 ==> [34, 54, 65]
    jshell> numbers.subSet(34, false, 65, false);
    $11 ==> [54]
```

`headSet()` returns the subset of elements preceding the given element value. `tailsSet()` returns the subset of elements succeeding the given element value

```
jshell> numbers.headSet(50);
$12 ==> [12, 34]
jshell> numbers.tailSet(50);
$13 ==> [54, 65, 99]
jshell>
```

The `Queue` Interface

`Queue` `interface` : `extends` the `Collection` `interface` .

- Elements are arranged in order of processing, such as in a To-Do List.

The `PriorityQueue` Collection

The `PriorityQueue` collection is a built-in Java `class` , that `implements` the `Queue` `interface` . Elements are stored in a sorted natural order, by default. We can also specify a different custom order, called the *order of priority* (customized by the programmer).

Snippet-17 : PriorityQueue

The `PriorityQueue` `queue` stores the strings in ascending alphabetic order by default.

- `queue.poll()` de-queue's the element at the beginning of the natural order

```
jshell> Queue<String> queue = new PriorityQueue<>();
numbers ==> [12, 34, 54, 65, 99]
jshell> queue.poll();
$1 ==> null
jshell> queue.offer("Apple");
$2 ==> true
jshell> queue.addAll(List.of("Zebra", "Monkey", "Cat"));
$3 ==> true
jshell> queue
queue ==> [Apple, Cat, Monkey, Zebra]
jshell> queue.poll();
$4 ==> "Apple"
jshell> queue
queue ==> [Cat, Monkey, Zebra]
jshell> queue.poll();
$5 ==> "Cat"
jshell> queue.poll();
$6 ==> "Monkey"
jshell> queue.poll();
$7 ==> "Zebra"
jshell> queue.poll();
$8 ==> null
jshell>
```

Snippet-18 : Custom Priority PriorityQueue

A custom priority order on the elements in a PriorityQueue can be specified by passing an implementation of `Comparator<? super T>` `interface` to the `PriorityQueue<T>` constructor.

Let's implement a `StringLengthComparator` .

QueueRunner.java

```
package collections;
import java.util.Comparator;
import java.util.List;
import java.util.Queue;
import java.util.PriorityQueue;

class StringLengthComparator implements Comparator<String> {
    @Override
    public int compare(String left, String right) {
        return Integer.compare(left.length(), right.length());
    }
}

public class QueueRunner {
    public static void main(String[] args) {
        Queue<String> queue = new PriorityQueue<>(new StringLengthComparator());
        queue.addAll(List.of("Zebra", "Monkey", "Cat"));
        queue.poll();
        queue.poll();
        queue.poll();
        queue.poll();
    }
}
```

Console Output

Cat

Zebra

Monkey

null

The `Map` `interface`

The `Map` `interface` specifies a contract to implement collections of elements, that are in the form of `(key, value)` pairs.

Let's say you want to store how many times a character is repeated in a sequence of characters.

- If the sequence inserted is: `{'A', 'C', 'A', 'C', 'E', 'C', 'M', 'D', 'H', 'A'}`
- The map contents would end up being: `{('A', 3), ('C', 3), ('E', 1), ('M', 1), ('D', 1), ('H', 1)}` .

Since the kind of elements stored in a map (`(key, value)` pairs) are different from any other collection categories in Java, the `Map` `interface` is unique. So unique, that it even does not extend the `Collection` `interface` !

The `interface` definition looks something like:

```
public interface Map<K, V> {

    //Method Declarations

}
```

- The Java collection classes that `implement` the `Map` `interface` are:
    - `HashMap`
    - `HashTable`
    - `LinkedHashMap`
    - `TreeMap`

`Map` Collections : Concepts

- `HashMap`

    - Unordered
    - Unsorted
    - Key's `hashCode()` value is used in the hashing function
    - Allows a key with a `null` value.

- `HashTable`

    - Thread-safe version of `HashMap` . Has `synchronized` methods where required.
    - Unordered
    - Unsorted
    - Key's `hashCode()` value is used in the hashing function
    - Does not allow a `null` key.

- `LinkedHashMap`

    - Insertion order of elements is maintained (which is optional as well)
    - Unsorted
    - Iteration is faster
    - Insertion and Deletion are slower

- `TreeMap`

    - Additionally implements the `NavigableMap` `interface`
    - Elements are maintained in sorted order of keys

`Map` `interface` : Basic Operations

Snippet-19 : Basic Map Operations

The `Map.of()` method takes a sequence of objects, which are interpreted as being key and value entries in alternation.

- The total number of arguments is expected to be an even number.
- This method creates an immutable `Map` , with the `(key, value)` pairs in no specific order.

```
jshell> Map<String, Integer> map = Map.of("A", 3, "B", 5, "Z", 10);
map ==> {Z=10, A=3, B=5}
jshell> map.get("Z");
$1 ==> 10
jshell> map.get("A");
$2 ==> 3
jshell> map.get("C");
$3 ==> null
jshell> map.size();
$4 ==> 3
jshell> map.isEmpty();
$5 ==> false
jshell> map.containsKey("A");
$6 ==> true
jshell> map.containsKey("F");
$7 ==> false
jshell> map.containsValue(3);
$8 ==> true
jshell> map.containsValue(4);
$9 ==> false
jshell> map.keySet();
$10 ==> [Z, A, B]
jshell> map.values();
$11 ==> [10, 3, 5]
jshell>
```

Snippet-20 : Mutable Maps

To be able to add values to a map, let's create a `HashMap` .

```
jshell> Map<String, Integer> map = Map.of("A", 3, "B", 5, "Z", 10);
map ==> {Z=10, A=3, B=5}
jshell> Map<String, Integer> hashMap = new HashMap<>(map);
hashMap ==> {A=3, Z=10, B=5}
jshell> hashMap.put("F", 5);
$1 ==> null
jshell> hashMap
hashMap ==> {A=3, Z=10, B=5, F=5}
jshell> hashMap.put("Z", 11);
$2 ==> 10
jshell> hashMap
hashMap ==> {A=3, Z=11, B=5, F=5}
jshell>
```

`Map` Collections: Comparing Operations

Snippet-21 : `Map` Implementations

`HashMap` elements are not guaranteed to be stored either in inserted order, or in the sorted order of keys.

```
jshell> HashMap<String, Integer> hashMap = new HashMap<>();
hashMap ==> {}
jshell> hashMap.put("Z", 5);
$1 ==> null
jshell> hashMap.put("A", 15);
$2 ==> null
jshell> hashMap.put("F", 25);
$3 ==> null
jshell> hashMap.put("L", 250);
$4 ==> null
jshell> hashMap
hashMap ==> {A=15, F=25, Z=5, L=250}
```

`LinkedHashMap` elements are stored in inserted order.

```
jshell> LinkedHashMap<String, Integer> linkedHashMap = new LinkedHashMap<>();
linkedHashMap ==> {}
jshell> linkedHashMap.put("Z", 5);
$5 ==> null
jshell> linkedHashMap.put("A", 15);
$6 ==> null
jshell> linkedHashMap.put("F", 25);
$7 ==> null
jshell> linkedHashMap.put("L", 250);
$8 ==> null
jshell> linkedHashMap
hashMap ==> {Z=5, A=15, F=25, L=250}
```

`TreeMap` elements are stored in the natural sorted order of the keys.

```
jshell> TreeMap<String, Integer> treeMap = new TreeMap<>();
treeMap ==> {}
jshell> treeMap.put("Z", 5);
$9 ==> null
jshell> treeMap.put("A", 15);
```

```
$10 ==> null
jshell> treeMap.put("F", 25);
$11 ==> null
jshell> treeMap.put("L", 250);
$12 ==> null
jshell> treeMap
treeMap ==> {A=15, F=25, L=250, Z=5}
jshell>
```

Exercise Set

1. Given the string: `"This is an awesome occassion. This has never happened before."` , do the following processing on it:
    - Find the number of occurrences of each unique character in this string
    - Find the number of occurrences of each unique word in this string

    Solution

MapRunner.java

```
package collections;
import java.util.Map;
import java.util.HashMap;

public class MapRunner {
    public static void main(String[] args) {
        String str = "This is an awesome occassion. This has never happened before.";
        char[] characters = str.toCharArray();
        Map<Character, Integer> occurrences = new HashMap<>();
        for(char character:characters) {
            Integer count = occurrences.get(character);
            if(count == null) {
                occurrences.put(character, 1);
            } else {
                occurrences.put(character, count+1);
            }
        }
        System.out.println(occurrences);
        String words = text.split(" ");
        Map<String, Integer> frequency = new HashMap<>();
        for(String word:words) {
            Integer number = frequency.get(word);
            if(number == null) {
                frequency.put(word, 1);
            } else {
                frequency.put(word, number+1);
            }
        }
        System.out.println(frequency);
    }
}
```

Revisiting `TreeMap`

Let's look at some more interesting operations of Data Structures based on `TreeMap` .

Snippet-13 : Treemap Operations

Entries in a `TreeMap` are always sorted.

```
jshell> TreeMap<String, Integer> treeMap = new TreeMap<>();
```

```
treeMap ==> {}
jshell> treeMap.put("F", 25);
$1 ==> null
jshell> treeMap.put("Z", 5);
$2 ==> null
jshell> treeMap.put("L", 250);
$3 ==> null
jshell> treeMap.put("A", 15);
$4 ==> null
jshell> treeMap.put("B", 25)
$5 ==> null
jshell> treeMap.put("G", 25);
$6 ==> null
jshell> treeMap
treeMap ==> {A=15, B=25, F=25, G=25, L=250, Z=5}
```

`TreeMap` implements `NavigableMap` `interface` as well.

```
jshell> treeMap.higherKey("B");
$7 ==> "F"
jshell> treeMap.higherKey("C");
$8 ==> "F"
jshell> treeMap.ceilingKey("B");
$9 ==> "B"
jshell> treeMap.ceilingKey("C");
$10 ==> "F"
jshell> treeMap.lowerKey("B");
$11 ==> "A"
jshell> treeMap.floorKey("B");
$12 ==> "B"
jshell> treeMap.firstEntry();
$713 ==> A=15
jshell> treeMap.lastentry();
$14 ==> Z=5
jshell> treeMap.subMap("C", "Y");
$7 ==> {F=25, G=25, L=250}
jshell> treeMap.subMap("B", "Z");
$7 ==> {B=25, F=25, G=25, L=250}
jshell> treeMap.subMap("B", true, "Z", true);
$7 ==> {B=25, F=25, G=25, L=250, Z=5}
jshell>
```

Java Collections : Conclusions With Tips

The collection interfaces we have explored, as well as their implementation classes, are:

- `List`
- `Set`
- `Queue`
- `Map`

Let's quickly review what we've learnt:

- When you use a "Hashed" Java collection (hash table based), such as `HashMap` or `HashSet` , it will be unordered, unsorted and will iterate over its elements in no particular order.
- When you encounter a "Linked" Java collection (linked list based), such as a `LinkedHashSet` or a `LinkedHashMap` , it will maintain insertion order, but will be unsorted.
- When we make use of a "Tree"-based Java collection (stored in a tree data structure), such as `TreeSet` or `TreeMap` it always maintains natural sorted order. It would implement one of the navigable category of interfaces, such as `NavigableSet` or `NavigableMap` .

Introducing Generics

Recommended Videos

- Generics - https://www.youtube.com/watch?v=v4o0wyFPwEs

Why do we need Generics?

Let's look at a scenario where want to write a wrapper class around the `ArrayList` data structure, maybe to do some better custom error-checking and stuff. For now, we will just look at basic wrapper functionality, the error checking intent is just an excuse!

Snippet-1

GenericsRunner.java

```
package com.in28minutes.generics;
import com.in28minutes.generics.MyCustomList;

public class GenericsRunner {
    public static void main(String[] args) {
    MyCustomList list = new MyCustomList();
    list.addElement("Element-1");
    list.addElement("Element-2");
    list.addElement("Element-3");
    }
}
```

MyCustomList.java

```
package com.in28minutes.generics;

public class MyCustomList {
    ArrayList<String> list = new ArrayList<>();

    public void addElement(String element) {
        list.add(element);
    }

    public void removeElement(String element) {
        list.remove(element);
    }
}
```

Snippet-1 Explained

The `MyCustomList` `class` is a wrapper for `ArrayList` of `String` s. Insertion and deletion of elements into this data structure is straightforward.

Let's sy I would want to create `MyCustomList` for other types. Should we write additional wrapper classes `MyCustomList` and so on?

Let's look at an example:

Snippet-2 : Implementing a Generic

MyCustomList.java

```java
package com.in28minutes.generics;

public class MyCustomList<T> {
    ArrayList<T> list = new ArrayList<>();

    public void addElement(T element) {
        list.add(element);
    }

    public void removeElement(T element) {
        list.remove(element);
    }

    public String toString() {
        return list.toString();
    }
}
```

GenericsRunner.java

```java
package com.in28minutes.generics;
import com.in28minutes.generics.MyCustomList;

public class GenericsRunner {
    public static void main(String[] args) {
        MyCustomList<String> list = new MyCustomList<>();
        list.addElement("Element-1");
        list.addElement("Element-2");
        System.out.println(list);

        MyCustomList<Integer> list2 = new MyCustomList<>();
        list2.addElement(Integer.valueOf(5));
        list2.addElement(Integer.valueOf(9));
        System.out.println(list2);
    }
}
```

Console Output

[Element-1, Element-2]

[5, 9]

Snippet-2 Explained

The identifier `T` in the definition of the Generic `class` `MyCustomList<T>` is a placeholder for the actual type of the container. It is this placeholder that truly converts the `MyCustomList` `class` into a template.

The naming convention for these type placeholders is: * Always use UpperCase letters of the alphabet (such as `T`, or `S`), or * intuitive words such as `TYPE`

At the time of actual instantiation of `MyCustomList` inside `GenericsRunner.main`, this placeholder is substituted by the actual type:

- When `MyCustomList<String> list` is created, `T` is substituted by `String`
- When `MyCustomList<Integer> list2` is created, `T` is substituted by `Integer`

Exercise Set - 18

1. Write a method `get` inside the generic class `MyCustomList`, which returns the element at a particular index (passed as argument) in its list storage.

Solution

MyCustomList.java

```java
package com.in28minutes.generics;

public class MyCustomList<T> {
    ArrayList<T> list = new ArrayList<>();

    public void addElement(T element) {
        list.add(element);
    }

    public void removeElement(T element) {
        list.remove(element);
    }

    public String toString() {
        return list.toString();
    }

    public T get(int index) {
        return list.get(index);
    }
}
```

GenericsRunner.java

```java
package com.in28minutes.generics;
import com.in28minutes.generics.MyCustomList;

public class GenericsRunner {
    public static void main(String[] args) {
        MyCustomList<String> list = new MyCustomList<>();
        list.addElement("Element-1");
        list.addElement("Element-2");
        String text = list.get(0);
        System.out.println(text);

        MyCustomList<Integer> list2 = new MyCustomList<>();
        list2.addElement(Integer.valueOf(5));
        list2.addElement(Integer.valueOf(9));
        Integer num = list2.get(1);
        System.out.println(num);
    }
}
```

Console Output

Element-1

9

Solution Explained

We have defined a method `MyCustomList<T>.get` whose return type is generic as well. The return type has the same placeholder `T` as the template in the definition of `MyCustomList<T>` .

- For `MyCustomList<String> list` , `list.get` returns a `String`
- For `MyCustomList<Integer> list2` , `list2.get` returns an `Integer`

Implementing Type Restrictions on Generics

We saw above that we could use `MyCustomList<T>` to be instantiated into data structures for storing `String` s as well as for `Integer` s.

What if we wanted to to use `MyCustomList<T>` purely for storing numeric values?

Snippet-3 : Generic Type Restrictions

MyCustomList.java

```
package com.in28minutes.generics;

public class MyCustomList<T extends Number> {
    ArrayList<T> list = new ArrayList<>();

    public void addElement(T element) {
        list.add(element);
    }

    public void removeElement(T element) {
        list.remove(element);
    }

    public String toString() {
        return list.toString();
    }

    public T get(int index) {
        return list.get(index);
    }
}
```

GenericsRunner.java

```
package com.in28minutes.generics;
import com.in28minutes.generics.MyCustomList;

public class GenericsRunner {
    public static void main(String[] args) {
        //MyCustomList<String> list = new MyCustomList<>();
        MyCustomList<Long> list1 = new MyCustomList<>();
        list1.addElement(5l);
        list1.addElement(7l);
        Long long = list1.get(0);
        System.out.println(long);

        MyCustomList<Integer> list2 = new MyCustomList<>();
        list2.addElement(Integer.valueOf(5));
        list2.addElement(Integer.valueOf(9));
        Integer num = list2.get(1);
        System.out.println(num);
    }
}
```

Console Output

5

9

When we specify `T extends Number` as the type, we can use all the methods in the API of `class Number` are available for use.

Generic Methods

We can create generic methods as well. Let's look at a few examples:

Snippet-4 : Generic Method

GenericsRunner.java

```
package com.in28minutes.generics;
import com.in28minutes.generics.MyCustomList;

public class GenericsRunner {
    static <X> X doSomething(X value) {
        return value;
    }

    static <X extends List> void duplicate(X list) {
        list.add(list);
    }

    public static void main(String[] args) {
        String text = doSomething("Hello");
        Integer value = doSomething(Integer.valueOf(7));
        ArrayList<String> list = doSomething(new ArrayList<String>(List.of("A", "B", C")));
        duplicate(list);
        System.out.println(list);
        LinkedList<Integer> list2 = doSomething(new LinkedList<String>(List.of(1, 2, 3)));
        duplicate(list2);
        System.out.println(list2);
    }
}
```

Console Output

[A, B, C, A, B, C]

[1, 2, 3, 1, 2, 3]

Generics And Wild-Cards

You can use wild card with generics too - `? extends Number`

Snippet-5

GenericsRunner.java

```
package com.in28minutes.generics;
import com.in28minutes.generics.MyCustomList;

public class GenericsRunner {
    static double sumOfNumberList(List<? extends Number> numbers) {
        double sum = 0.0;
        for(Number number:numbers) {
            sum += number;
        }
        return sum;
    }

    public static void main(String[] args) {
        System.out.println(sumOfNumberList(List.of(1, 2, 3, 4, 5)));
        System.out.println(sumOfNumberList(List.of(1.1, 2.1, 3.1, 4.1, 5.1)));
        System.out.println(sumOfNumberList(List.of(1l, 2l, 3l, 4l, 5l)));
        }
    }
```

Console Output

15.0

15.5

15.0

Snippet-5 Explained

The symbol `?` in the definition of the method `static double sumOfNumberList(List<? extends Number> numbers)` is the **wild-card** symbol. It denotes the fact that in order to be a valid argument to `sumOfNumberList`, `numbers` can be a `List` of any elements, so long as all of them are of type sub-classed from `Number`.

- This includes `Integer`, `Long`, `Short`, `Byte`, `Float` and `Double`.
- It also includes their primitive type counterparts, since they can be converted implicitly to their Wrapper class counterparts.
- Of course, all these elements of `List` `numbers` need to be of a homogeneous type.

Restricted Heterogeneous Lists

The generic wildcard we saw in the previous section is referred to as a **Upper-Bounded Wild-Card**. It can be used to specify homogeneous types with a restriction. There is another category of wild-cards called **Lower-Bounded Wild-Card**, which can be used with create **Heterogeneous** types of elements , within the restriction. Here is an example.

Snippet-6 : More wild-cards

GenericsRunner.java

```
    package com.in28minutes.generics;
    import com.in28minutes.generics.MyCustomList;

    public class GenericsRunner {
        static void addAFewNumbers(List<? super Number> numbers) {
            numbers.add(1);
            numbers.add(1l);
            numbers.add(1.0);
            numbers.add(1.0l);
        }

        public static void main(String[] args) {
            List<Number> numberList = new ArrayList<>();
            addAFewNumbers(numberList);
            System.out.println(numberList);
        }
    }
```

Console Output

[1, 1, 1.0. 1.0]

Introduction to Functional Programming

What's all the fuss around Functional Programming about?

Let's find out.

Functional Programming Videos

- Part 1 - https://www.youtube.com/watch?v=aFCNPHfvqEU
- Part 2 - https://www.youtube.com/watch?v=5Xw1_IREXQs

Step 01: Introducing Functional Programming

Let's look at a typical program to loop around a list and print its content.

Snippet-01 : OOP List Traversal

FunctionalProgrammingRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class FunctionalProgrammingRunner {
    public static void main(String[] args) {
        List<String> list = List.of("Apple", "Banana", "Cat", "Dog");
        for(String str:list) {
            System.out.println(str);
        }
    }
}
```

Console Output

Apple

Banana

Cat

Dog

Snippet-01 Explained

Above approach focuses on the **how**.

We looped around the list, accessed individual elements of a `List` and did `System.out.println()` to print each element.

Functional Programming allows us to focus on the **what**.

Snippet-02 : `printBasic()` And `printFunctional()`

FunctionalProgrammingRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class FunctionalProgrammingRunner {
    public static void main(String[] args) {
        List<String> list = List.of("Apple", "Banana", "Cat", "Dog");
        //printBasic(list);
        printFunctional(list);
    }

    public static void printBasic(List<String> list) {
        for(String str:list) {
            System.out.println(str);
        }
    }

    public static void printFunctional(List<String> list) {
        list.stream().forEach(
                                element -> System.out.println(element)
                             );
    }
}
```

Console Output

Apple

_Banana_Cat_

Dog

Snippet-02 Explained

`list.stream().forEach(element -> System.out.println(element))` - for each element in list stream, print it.

`element -> System.out.println(element)` is called a **lambda expression**.

Step 02: Looping Through A `List`

In the previous step , we use this snippet of code - `list.stream().forEach(element -> System.out.println(element))`

We are **"Passing a function as a method argument"**.

Let's use JShell to explore this further.

Let's try to print a list of numbers.

Snippet-01 : Loop Using FP

```
    jshell> List<Integer> list = List.of(1, 4, 7, 9);
    list ==> [1, 4, 7, 9]
    jshell> list.stream().forEach(elem -> System.out.println(elem));
    1
    4
    7
    9
    jshell>
```

Snippet-01 Explained

`elem -> System.out.println(elem)` is a lambda expression. For each element in list stream, execute the lambda expression.

Step 03: Filtering Results

A `Stream` is a sequence of values. The `filter()` method can be used to filter the `Stream` elements based on some logic.

Snippet-01 : Using `filter()`

`printBasicWithFiltering` shows the usual approach of filtering. `printFPWithFiltering` shows the functional approach.

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class FunctionalProgrammingRunner {
    public static void main(String[] args) {
        List<String> list = List.of("Apple", "Bat", "Cat", "Dog");
        //printBasicWithFiltering(list);
        printFPWithFiltering(list);
    }

    public static void printBasicWithFiltering(List<String> list) {
        for(String str:list) {
            if(str.endsWith("at")) {
                System.out.println(str);
            }
        }
    }

    public static void printFPWithFiltering(List<String> list) {
        list.stream()
            .filter(elem -> elem.endsWith("at"))
            .forEach(element -> System.out.println(element));

    }
}
```

Console Output

Bat

Cat

Snippet-02 : Printing even/odd numbers

Let's look at how to filter numbers.

```
jshell> List<Integer> list = List.of(1, 4, 7, 9);
list ==> [1, 4, 7, 9]
jshell> list.stream().forEach(elem -> System.out.println(elem));
1
4
7
9
jshell> list.stream().filter(num -> num%2 == 1).forEach(elem -> System.out.println(elem));
1
7
9
jshell> list.stream().filter(num -> num%2 == 0).forEach(elem -> System.out.println(elem));
4
jshell>
```

Snippet-02 Explained

Typically, these are the conditions we write

- `num` is odd : `if(num % 2 == 1) { /* */ }`
- `num` is even : `if(num % 2 == 0){ /* */ }`

In the above example, we are using lambda expression to define the same conditions.

- `num` is odd: `num -> num%2 == 1`

- `num` is even: `num -> num%2 == 0`

Step 05: Streams - Aggregated Results

Sometimes we want to aggregate data into a single result. For example, we might want to add all the numbers between `1` and `10`. Or we may want to calculate the average maximum temperature in our city over a month's time.

Snippet-01 : Sum Of A Sequence

Let's look at how to use `reduce` method to calculation the sum. ***FPNumberRunner.java***

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class FPNumberRunner {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(4, 6, 8, 13, 3, 15);
        //System.out.println(printBasicSum(numbers));
        int sum = numbers.stream()
                         .reduce(
                                 0,
                                 (num1, num2) -> num1 + num2
                                );
        System.out.println(sum);
    }

    void printBasicSum(List<Integer> numbers) {
        int sum=0;
        for(int num:numbers) {
            sum += num;
        }
        System.out.println(sum);
    }
}
```

Console Output

49

Snippet-01 Explained

The `reduce()` method acts on a pair of elements at a time. The *initial-value* is `0`. The lambda expression `(num1, num2) -> num1 + num2` is executed on the elements of the list, a pair at a time.

Classroom Exercise CE-01

1. Given the list `(4, 6, 8, 13, 3, 15)`, compute:
   - The sum of even numbers in the list.
   - The sum of odd numbers in the list.

Solution To CE-01

FPNumberRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class FPNumberRunner {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(4, 6, 8, 13, 3, 15);
        printFPEvenSum(numbers);
        printFPOddSum(numbers);
    }

    void printFPEvenSum(List<Integer> numbers) {
        int sum = numbers.stream()
                         .filter(elem -> elem %2 == 0)
                         .reduce(0, (num1, num2) -> num1 + num2);
        System.out.println("Even Numbers Sum: " + sum);
    }

    void printFPOddSum(List<Integer> numbers) {
        int sum = numbers.stream()
                         .filter(elem -> elem %2 == 1)
                         .reduce(0, (num1, num2) -> num1 + num2);
        System.out.println("Odd Numbers Sum: " + sum);
    }
}
```

Console Output

Even Numbers Sum: 18

Odd Numbers Sum: 31

Step 06: Functional Programming v Structured Programming

Let's have a re-look at the ***FPNumberRunner.java*** program from the previous step. We wrote two variants of the same task that computed the sum of a list of numbers:

- `basicSum()` : that used the traditional approach
- ```fpSum()``: which followed the *FP* scheme of things

Let's use them as benchmarks, to illustrate the core differences between *SP* and *FP*.

1. **Structured Programming** (**SP**)

```
    public int basicSum(List<Integer> numbers) {
        int sum=0;
        for(int num:numbers) {
            sum += num;
        }
        return sum;
    }
```

2. **Functional Programming** (**FP**)

```
    public int  fpSum(List<Integer> numbers) {
        int sum = numbers.stream()
                         .reduce(0, (num1, num2) -> num1 + num2);
        return sum;
    }
```

How are these different?

- **Mutations** (changes to program data):

    i. *SP*: Within `basicSum()` , the variable `sum` (a sort of worker variable) is initialized to `0` , and undergoes *mutations* across iterations of the `for` loop.

ii. *FP*: We just set up an initial value for `reduce()` to work with. We don't have any mutation.

- The **what** and **how** of computation:

  i. *SP*: We specify both *what* to do, and *how* to do it. The code loops through the list elements using a `for`, while also updating the aggregate value in `sum`.

  ii. *FP*: We are focused on *what* to do, and very little on the *how* part. `reduce()` takes care of what numbers from the *stream* to add up, and you don't bother about how to select them from the *stream*.

Step 07: Some FP Terminology

Let's look at some *FP* terminology a little more formally.

Lambda Expression

```
(num1, num2) -> num1 + num2
```

is equivalent of this method

```
int basicSum(int num1, int num2) {
    return num1 + num2;
}
```

A lambda expression can have multiple lines of Java code as well:

```
(num1, num2) -> {
    System.out.println(num1 + " " + num2);
    return num1 + num2;
}
```

Why take our word for all this? Let's put this code into an IDE, and then run it, to see for ourselves.

Snippet-01 : Lambda Expression

FPNumberRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class FPNumberRunner {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(4, 6, 8, 13, 3, 15);
        System.out.println(numbers);
        printFPSum(numbers);
    }

    void printFPSum(List<Integer> numbers) {
        int sum = numbers.stream()
                         .reduce(0,
                                 (num1, num2) -> {
                                                   System.out.println(num1 + " " + num2);
                                                   return num1 + num2;
                                                 }
        System.out.println("Even Numbers Sum: " + sum);
    }
}
```

Console Output

[4, 6, 8, 13, 3, 15]

0 4

4 6

10 8

18 13

31 3

34 15

49

Stream

A Stream is a sequence of elements. You can perform different kinds of operations on a stream.

- **Intermediate Operations**: An operation that *takes a stream* - for example, applies a lambda expression - and produces *another stream* of elements as its result.
- **Terminal Operations**: A stream operation that takes a stream - for example, applies a lambda expression - and returns a single result (A single primitive-value/object, or a single collection-of-objects). `reduce()` and `forEach()` are a couple of such operations.

Step 08: Intermediate Stream Operations

Output of an intermediate stream operation is another stream.

The most popular intermediate stream operations are:

- `sorted()`
- `distinct()`
- `filter()`
- `map()`

In this step, let's check them out one by one.

Snippet-01 : `sorted()` and other operations

```
    jshell> List<Integer> numbers = List.of(3, 5, 8, 213, 45, 4, 7);
    numbers ==> [3, 5, 8, 213, 45, 4, 7]
    jshell> numbers.stream().sorted().forEach(elem -> System.out.println(elem));
    3
    4
    5
    7
    8
    45
    213
    jshell> List<Integer> numbers = List.of(3, 5, 3, 213, 45, 5, 7);
    numbers ==> [3, 5, 3, 213, 45, 5, 7]
    jshell> numbers.stream().distinct().forEach(elem -> System.out.println(elem));
    3
    5
    213
    45
    7
    jshell> numbers.stream().distinct().sorted().forEach(elem -> System.out.println(elem));
```

```
3
5
7
45
213
jshell> numbers.stream().distinct().map(num -> num*num).forEach(elem -> System.out.println(elem)
9
25
45369
2025
49
jshell>
```

Snippet-01 Explained

- `sorted()` preserves the elements of the consumed stream in the result, but also puts them in natural sorted order (Increasing order for numbers, alphabetical order for strings).
- `distinct()` returns a stream retaining only the unique elements of the input stream. This method maintains the relative order of retained elements.
- You can chain together more than one intermediate operation, such as `sorted()` followed by `distinct()` above. Such code is sometimes called a *pipeline*.
- `map()` : Applies a lambda expression to compute new results from the input stream elements. It then returns a stream of these results as output. In our example, `map()` takes each element in the `Stream` object created by ```number.stream()``,` to its square value.

Step 09: Programming Exercise FP-PE-01

Exercises

1. Write a program to print the squares of the first 10 positive integers.
2. Create a list of the character strings "Apple", "Banana" and "Cat". Print all of them in lower-case.
3. Create a list of the character strings "Apple", "Banana" and "Cat". Print the length of each string.

Solutions To FP-PE-01

FPNumberRunner.java

```
package com.in28minutes.functionalprogramming;
public class FPNumberRunner {
    public static void printFPSquares() {
        IntStream.range(1, 11).
                  map(num -> num*num).
                  forEach(elem -> System.out.println(elem));
    }

    public static void printLowerCases(List<String> list) {
    }

    public static void printLengths(List<String> list) {
        list.stream()
            .map(s -> s.length())
            .forEach(elem -> System.out.println(elem);
    }

    public static void main(String[] args) {
        printFPSquares();
        List<String> list = List.of("Apple", "Banana", "Cat");
        printLowerCases(list);
        printLengths(list);
```

```
        }
    }
```

Console Output

1

4

9

16

25

36

49

64

81

100

apple

banana

cat

5

6

3

Solution Explained

- The `map()` method accepts a lambda expression.

Step 10: Terminal Operations

A terminal operation returns a single result (A single object/data-unit, or a single collection). It does not return an output stream.

Commonly used instances of this are:

- `reduce()`
- `max()` and `min()`

```
jshell> IntStream.range(1, 11).reduce(0, (n1, n2) -> n1 + n2);
$1 ==> 55
```

`max()` expects a lambda expression providing a `Comparator<T>` implementation. `Integer.compare(n1,n2)` is an implementation of the `Comparator<T>` interface for comparing integers.

What if there are no numbers in the Stream? What should be returned? As a Java programmer, we grew up to hate `null`. You don't want to return `null` back.

The `Optional` type provides an alternative:

- You can query an `Optional` object to check if it contains a valid result, by invoking `isPresent()` on it.
- You can get that result by calling `get()` on the same object.

```
jshell> List.of(23, 12, 34, 53).stream().max();
| Error:
| method max() in interface java.util.stream.Stream<T> cannot be applied to given types
| required : java.lang.Comparator<? super java.lang.Integer>
| found : no argument
| List.of(23, 12, 34, 53).stream().max();
|^-----------------------------------^
jshell> List.of(23, 12, 34, 53).stream().max((n1, n2) -> Integer.compare(n1, n2));
$2 ==> Optional[53]
jshell> $2.isPresent()
$3 ==> true
jshell> List.of(23, 12, 34, 53).stream().max((n1, n2) -> Integer.compare(n1, n2)).get();
$4 ==> 53
jshell>
```

Step 11: More Stream Operations

Let's now play around with a few more terminal operations, such as `min()` (terminal operation), and `filter()` (intermediate operation).

Snippet-01 : min() and max()

Using `min()` is similar to `max()` .

```
jshell> List.of(23, 12, 34, 53).stream().max((n1, n2) -> Integer.compare(n1, n2)).get()
$1 ==> 53
jshell> List.of(23, 12, 34, 53).stream().min((n1, n2) -> Integer.compare(n1, n2)).get()
$2 ==> 12
jshell>
```

Snippet-02 : Odd and Even Numbers

`collect()` method can be called to collect the result of `filter()` into a list. `Collectors.toList()` is the utility method used.

```
jshell> List.of(23, 12, 34, 53).stream().filter(e -> e%2==1).forEach(e ->     System.out.printl
23
53
jshell> List.of(23, 12, 34, 53).stream().filter(e -> e%2==1).collect(Collectors.toList());
$1 ==> [23, 53]
jshell>
```

Classroom Exercise FP-CE-02

1. From a list of 23, 12, 34, 53, create a list of the even numbers in it.
2. Create a list of squares of the first 10 positive integers.

Solutions To FP-CE-02

FunctionalProgrammingRunner.java

```
package com.in28minutes.functionalprogramming;

public class FPNumberRunner {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(23, 12, 34, 53);
        List<Integer> evens = numbers.stream()
                                     .filter(n -> n%2==0)
                                     .collect(Collectors.toList());

        List<Integer> tenSquares = IntStream.range(1, 11)
                                            .map(n -> n*n)
                                            .boxed()
                                            .collect(Collectors.toList());
    }
}
```

Solution Explained

- Solution to #1 is straightforward, given that we already know how to use `filter()` .
- Solution #2 uses `boxed()` method to convert an `IntPipeline` to a `Stream` . After that, what follows is routine stuff.

Step 12: The `Optional<T>` `class`

In an earlier section, we wrote and ran the following code:

```
jshell> List.of(23, 12, 34, 53).stream().max((n1, n2) -> Integer.compare(n1, n2));
Optional[53]
jshell>
```

In order to get the result in a form you would appreciate, we modified this code to look like:

```
jshell> List.of(23, 12, 34, 53).stream().max((n1, n2) -> Integer.compare(n1, n2)).get();
53
jshell>
```

`max()` is a stream operation, that needs to consume a stream. It is possible that the input stream is empty. In that case, the maximum value would be `null` . It is undesirable in *FP* to encounter an exception during a stream operation. It is extremely inelegant if we are asked to handle an exception in an *FP* code pipeline.

The `Optional<T>` `class` was introduced in Java SE 8, as a lifeline in such situations. It covers the possibility of absence of (or `null`) result from a terminal stream operation. The following example illustrates how you can query, and access the result from, an `Optional<T>` object.

```
jshell> List.of(23, 45, 67, 12).stream().filter(num -> num % 2 == 0).max( (n1, n2) ->     Intege
$1 ==> Optional[12]
jshell> $1.get()
$2 ==> 12
jshell> $1.isPresent()
$3 ==> true
```

In case the result is empty, then the value stored in the result is not `null` , it is `Optional.empty` .

```
jshell> List.of(23, 45, 67).stream().filter(num -> num % 2 == 0).max( (n1, n2) ->     Integer.cor
$4 ==> Optional.empty
jshell> $4.isPresent()
$5 ==> false
jshell> $4.orElse(0)
$6 ==> 0
```

You can provide a default value for the result using the method `orElse()` .

```
jshell> List.of(23, 45, 67).stream().filter(num -> num % 2 == 0).max( (n1, n2) ->     Integer.cor
$7 ==> 0
jshell> List.of(23, 45, 67, 34).stream().filter(num -> num % 2 == 0).max( (n1, n2) ->     Integer
$8 ==> 34
jshell>
```

Step 13: Functional Interfaces : `Predicate`

When we define a lambda expression , a lot of things happen behind the scenes.

An important concept is a *functional interface*.

Let's explain this term using an example.

The following code takes a behind-the-scenes look at `filter()` .

Snippet-01: Lambda behind-the-scenes - v1

LambdaBehindTheScenesRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class LambdaBehindTheScenesRunner {
    public static void main(String[] args) {
        List.of(23, 43, 34, 45).stream()
                               .filter(num -> num%2 == 0)
                               .forEach(e -> System.out.println(e));
    }
}
```

Console Output

34

36

48

Snippet-01 Explained

The signature of `filter()` reads is `Stream<T> java.util.stream.Stream.filter(Predicate<? super T> predicate)` . In this case, `T` is `java.lang.Integer` .

`filter()` accepts an object implementing the `Predicate` interface, as its argument. It returns a stream, consisting of those elements from the input stream, that match this predicate.

Conventionally speaking, a predicate is a logical condition. This predicate is applied to each element, to determine if it should be included in the output stream.

The `Predicate<T> interface is` an example of a *Functional Interface*. This interface has one method `boolean test(T t)` .

Instead of `num -> num%2 == 0` , let's implement a `EvenNumberPredicate` .

```
@FunctionalInterface
public interface Predicate<? super T> {
    boolean test(T t) { /*  */ }
        //...
    }
}
```

Snippet-02: lambda behind the scenes - v2

LambdaBehindTheScenesRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;
import java.util.function.Predicate;

class EvenNumberPredicate implements Predicate<Integer> {
    @Override
    public boolean test(Integer number) {
        return (number%2 == 0);
    }
}

public class LambdaBehindTheScenesRunner {
    public static void main(String[] args) {
        List.of(23, 43, 34, 45, 36, 48).stream()
                                       .filter(new EvenNumberPredicate())
                                       .forEach(e -> System.out.println(e));
    }
}
```

Console Output

34

36

48

Snippet-02 Explained

`EvenNumberPredicate` implements the `Predicate<Integer>` interface, and overrides the `test()` method.

Something similar to `EvenNumberPredicate` is created when we use lambda expressions `num -> num%2 == 0` .

Step 14: Functional Interfaces : `Consumer`

Let's look at another Functional Interface - `Consumer<S>` .

`forEach()` on a stream is actually defined as - `forEach(Consumer<? super S> action)`

The `Consumer<S>` interface has the following definition:

```
@FunctionalInterface
public interface Consumer<? super S> {
    void accept(S s) { /*  */ }

    //...

}
```

The lambda expression used inside `forEach()` and other such stream operations, actually represent a `Consumer` implementation.

Snippet-01

Let's implement a `SysOutConsumer` .

FunctionalProgrammingRunner.java

```java
    package com.in28minutes.functionalprogramming;
    import java.util.List;
    import java.util.stream.Stream;
    import java.util.function.Consumer;
    import java.util.function.Predicate;

    class EvenNumberPredicate implements Predicate<Integer> {
        @Override
        public boolean test(Integer num) {
            return num%2 == 0;
        }
    }

    class SysOutConsumer implements Consumer<Integer> {
        @Override
        public void accept(Integer num) {
            System.out.println(num);
        }
    }

    public class FPNumberRunner {
        public static void main(String[] args) {
            List<Integer> numbers = List.of(23, 12, 34, 45, 36, 48);
            //List<Integer> evens = numbers.stream()
                                           .filter(n -> n%2==0)
                                           .collect(Collectors.toList());

            List<Integer> evensToo = numbers.stream()
                                            .filter(new EvenNumberPredicate())
                                            .collect(Collectors.toList());

            numbers.stream()
                   .filter(new EvenNumberPredicate())
                   .forEach(new SysOutConsumer());
        }
    }
```

Console Output

12

34

36

48

Snippet-01 Explained

- The code actually speaks for itself. The steps to customize a `Consumer<S>` implementation are quote simple, and straightforward.
- The final code that involves both a custom `Predicate<T>` , and a custom `Consumer<S>` is still quite compact and elegant!

Step 15: More Functional Interfaces

Let's now look at what happens behind the scenes, for the stream operation `map()` . Suppose we wanted to print out the squares of all even numbers in a given list.

Snippet-01 : `map()` Behind The Scenes - v1

```
package com.in28minutes.functionalprogramming;
import java.util.List;
import java.util.stream.Stream;
import java.util.function.Consumer;
import java.util.function.Predicate;

class EvenNumberPredicate implements Predicate<Integer> {
    @Override
    public boolean test(Integer num) {
        return num%2 == 0;
    }
}

class SysOutConsumer implements Consumer<Integer> {
    @Override
    public void accept(Integer num) {
        System.out.println(num);
    }
}

public class FPNumberRunner {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(23, 12, 34, 45, 36, 48);
        numbers.stream()
               .filter(new EvenNumberPredicate())
               .map(n -> n*n)
               .forEach(new SysOutPredicate());
    }
}
```

Console Output

1156

1296

2304

Snippet-01 Explained

The signature of the `map()` intermediate stream operation is :

```
<R> Stream<R> map(Function<?super T, ? extends R> mapper){}
```

`Function` is shown below.

```
@FunctionalInterface
```

```
public interface Function<T,R> {
    R apply(T t);
}
```

The method `apply()` accepts a `T` object as argument, and returns another object of type `R` . In effect, any `Function` implementation can map object of one type to another.

Snippet-02 : `map()` Behind The Scenes - v2

Let's implement a `NumberSquareMapper` .

```
package com.in28minutes.functionalprogramming;
import java.util.List;
import java.util.stream.Stream;
import java.util.function.Consumer;
import java.util.function.Predicate;
import java.util.function.Function;

class EvenNumberPredicate implements Predicate<Integer> {
    @Override
    public boolean test(Integer num) {
        return num%2 == 0;
    }
}

class SysOutConsumer implements Consumer<Integer> {
    @Override
    public void accept(Integer num) {
        System.out.println(num);
    }
}

class NumberSquareMapper implements Function<Integer, Integer> {
    @Override
    public Integer apply(Integer number) {
        return number * number;
    }
}

public class FPNumberRunner {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(23, 12, 34, 45, 36, 48);
        numbers.stream()
                .filter(num -> num%2 == 0)
                .map(n -> n*n)
                .forEach(e -> System.out.println(e));

        numbers.stream()
                .filter(new EvenNumberPredicate())
                .map(new NumberSquareMapper())
                .forEach(new SysOutPredicate());
    }
}
```

Console Output

1156

1296

2304

1156

1296

2304

Step 16: Introducing Method References

What is a method reference?

Let's look at an example.

Snippet-01: Method References - v1

MethodReferencesRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class MethodReferencesRunner {
    public static void main(String[] args)
        List.of("Ant", "Bat", "Cat", "Dog", "Elephant").stream()
                                                       .map(s -> s.length())
                                                       .forEach(l -> System.out.println(l));
    }
}
```

Console Output

3

3

3

3

8

Snippet-02 : Method References - v2

Method references make it easy to create lambda expressions.

`l -> System.out.println(l)` can be replaced with `System.out::println`.

MethodReferencesRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class MethodReferencesRunner {
    public static void main(String[] args) {
        List.of("Ant", "Bat", "Cat", "Dog", "Elephant").stream()
                                                       .map(s -> s.length())
                                                       .forEach(l -> System.out.println(l));

        List.of("Ant", "Bat", "Cat", "Dog", "Elephant").stream()
                                                       .map(s -> s.length())
                                                       .forEach(System.out::println);
    }
}
```

Snippet-03: Method References - v3

Let's define a static method `print` and use it using a method reference.

`MethodReferencesRunner::print` is same as `l -> MethodReferencesRunner.print(l)`

MethodReferencesRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class MethodReferencesRunner {
    public static void print(Integer number) {
        System.out.println(number);
    }

    public static void main(String[] args) {
        List.of("Ant", "Bat", "Cat", "Dog", "Elephant").stream()
                                                        .map(s -> s.length())
                                                        .forEach(l -> MethodReferencesRunner.prin

        List.of("Ant", "Bat", "Cat", "Dog", "Elephant").stream()
                                                        .map(s -> s.length())
                                                        .forEach(MethodReferencesRunner::print);
    }
}
```

Snippet-04 : Method References - v4

Instance method calls can also be replaced with their method references.

On a `String` , `String::length` is the same as `s -> s.length()` .

MethodReferencesRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class MethodReferencesRunner {
    public static void print(Integer number) {
        System.out.println(number);
    }

    public static void main(String[] args) {
        List.of("Ant", "Bat", "Cat", "Dog", "Elephant").stream()
                                                        .map(s -> s.length())
                                                        .forEach(MethodReferencesRunner::print);

        List.of("Ant", "Bat", "Cat", "Dog", "Elephant").stream()
                                                        .map(String::length)
                                                        .forEach(MethodReferencesRunner::print);
    }
}
```

Classroom Exercise FP-CE-03

1. Using method references, write java functional code to determine the maximum even number, in a given list of integers.

Solution To FP-CE-03

MethodReferencesRunner.java

```
package com.in28minutes.functionalprogramming;
import java.util.List;

public class MethodReferencesRunner {
    public static void print(Integer number) {
        System.out.println(number);
    }

    public static void main(String[] args) {
        int max = List.of(23, 45, 67, 34).stream()
                                        .filter(num -> num % 2 == 0)
                                        .max( (n1, n2) -> Integer.compare(n1, n2) )
                                        .orElse(0);

        System.out.println(max);
        int maximum = List.of(23, 45, 67, 34).stream()
                                            .filter(MethodReferencesRunner::isEven)
                                            .max(Integer::compare)
                                            .orElse(0);

        System.out.println(maximum);
    }

    public static booelan isEven(Integer number) {
        return (number %2 == 0);
    }
}
```

Console Output

34

34

Summary

In this step, we:

- Understood what is a method reference
- Learned that both built-in, and user defined class methods can be invoked using method references
- Observed that method references work for static and non-static methods

Step 17: FP - Functions As First-Class Citizens

Are functions first class citizens in Java?

Here are few questions to think about?

- Can you pass a function as an argument to a method?
- Can you assign a function to a variable?
- Can you obtain a function as a return value, from a method invocation?

Passing function as method argument

We looked at several examples of this earlier.

In the example below, `num -> num % 2 == 0` is passed to `filter` method.

```
int max = List.of(23, 45, 67, 34).stream()
                                .filter(num -> num % 2 == 0)
```

```
                                    .max( (n1, n2) -> Integer.compare(n1, n2) )
                                    .orElse(0);
```

Storing functions in reference variables

`evenPredicate` and `oddPredicate` represent functions.

```
package com.in28minutes.functionalprogramming;
import java.util.List;
import java.util.function.Predicate;

public class FPNumberRunner {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(23, 12, 34, 45, 36, 48);
        Predicate<? super Integer> evenPredicate = num -> num % 2 == 0;
        Predicate<? super Integer> oddPredicate = num -> num % 2 == 1;

        numbers.stream()
               .filter(evenPredicate)
               .map(n -> n*n)
               .forEach(e -> System.out.println(e));
    }
}
```

Console Output

1156

1296

2304

Returning functions from methods

`createEvenPredicate` and `createOddPredicate` are examples of methods returning functions.

```
import java.util.stream.Stream;
import java.util.function.Predicate;

public class FPNumberRunner {
    public static Predicate<? super Integer> createEvenPredicate() {
        return num -> num%2 == 0;
    }

    public static Predicate<? super Integer> createOddPredicate() {
        return num -> num%2 == 1;
    }

    public static void main(String[] args) {
        List<Integer> numbers = List.of(23, 12, 34, 45, 36, 48);
        //Predicate<? super Integer> evenPredicate = num -> num % 2 == 0;
        //Predicate<? super Integer> evenPredicate = createEvenPredicate();

        numbers.stream()
               //.filter(num -> num%2 == 0)
               //.filter(evenPredicate)
               .map(n -> n*n)
               .forEach(e -> System.out.println(e));
    }
}
```

Console Output

1156

1296

2304

Summary

In this step, we observed that the following is true for a function:

- It can be passed as a method argument
- It can be stored in a reference variable
- It can be returned from a method

Threads and Concurrency

So far, we've only seen programs that run like a single horse, running round a race course.

However, it often makes sense for a program's *main task* to be split into smaller ones (let's call them *sub-tasks*).

Imagine an ant-colony, where a large number of worker ants toil together, to complete what the Queen Ant tells them to do. Groups of ants that are part of separately tasks, work with a free hand.

The concept of **concurrency** in programming is very similar to what an ant colony is in nature.

Step 01: Concurrent Tasks: Extending `Thread`

In Java, you can run tasks in parallel using threads. Let's first write a simple program.

Snippet-1

ThreadBasicsRunner.java

```
public class ThreadBasicsRunner {
    public static void main(String[] args) {
        //Task1
        for(int i=101; i<=199; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\n Task1 Done");

        //Task2
        for(int i=201; i<=299; i++) {
            System.out.print(i + " ");
        }

        System.out.println("\n Task2 Done");
        //Task3
        for(int i=301; i<=399; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\n Task3 Done");

        System.out.println("Main Done");
    }
}
```

Console Output

101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199

Task1 Done

201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299

Task2 Done

301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399

Task3 Done

Main Done

Snippet-1 Explained

As you can see, the execution of all three `for` loops (that really are independent tasks) is sequential. This is how all our code so far has been running!

Thread Creation

There are two ways in which you can create a thread to represent a sub-task, within a program. They are:

- Define your own thread `class` to sub-class the **`Thread`** `class`.
- Define your own thread `class` to implement the **`Runnable`** `interface`.

In this step, we will focus on the first alternative.

Snippet-01 : A simple Java thread class

ThreadBasicsRunner.java

```java
class Task1 extends Thread {
    public void run() {
        System.out.println("Task1 Started ");
        for(int i=101; i<=199; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\nTask1 Done");
    }
}

public class ThreadBasicsRunner {
    public static void main(String[] args) {
        //Task1
        Task1 task1 = new Task1();
        task1.start();

        //Task2
        for(int i=201; i<=299; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\nTask2 Done");
        //Task3
        for(int i=301; i<=399; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\nTask3 Done");
        System.out.println("\nMain Done");
    }
}
```

Console Output

```
Task1 Started
201 101 202 102 203 103 204 104 105 205 206 106 207 107 208 108 209 109 210 110 211 111 212 112 213 1
Task2 Done
301 199 302
Task1 Done
303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 3
Task3 Done

Main Done
```

Snippet-01 Explained

We defined a `Task1 class` to denote our sub-task, with a `run()` method definition. However, when we create such a thread within our `main()` method, we don't seem to be invoking `run()` in any way! What's happening here?

A thread can be created and launched, by calling a generic method named `start()` . method. Calling `start()` will invoke the individual thread's `run()` method.

From the console output, we see that the output of *Task1* overlaps with those of tasks labeled *Task2* and *Task3*. *Task1* is running in parallel with main which is running (*Task2*, *Task3*).

Summary

In this step, we:

- Discovered how to define a thread by sub-classing `Thread`
- Demonstrated how to run a Thread

Step 02: Concurrent Tasks - Implementing Runnable

Snippet-01 : Implementing Runnable

In **Step 01**, we told you that there are two ways a thread could represent a sub-task, in a Java program. One was by sub-classing a `Thread` , and the other way is to implement `Runnable` . We saw the first way a short while ago, and it's time now to explore the second. The following example will show you how it's done.

ThreadBasicsRunner.java

```
class Task1 extends Thread {
    public void run() {
        System.out.println("Task1 Started ");
        for(int i=101; i<=199; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\nTask1 Done");
    }
}
```

```
class Task2 implements Runnable {
    @Override
    public void run() {
        System.out.println("Task2 Started ");
        for(int i=201; i<=299; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\nTask2 Done");
    }
}

public class ThreadBasicsRunner {
    public static void main(String[] args) {
        System.out.print("\nTask1 Kicked Off\n");
        Task1 task1 = new Task1();
        task1.start();

        System.out.print("\nTask2 Kicked Off\n");
        Task2 task2 = new Task2();
        Thread task2Thread = new Thread(task2);
        task2Thread.start();

        System.out.print("\nTask3 Kicked Off\n");
        for(int i=301; i<=399; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\nTask3 Done");

        System.out.println("\nMain Done");
    }
}
```

Console Output

```
Task1 Kicked Off
Task1 Started
101 102 103 104 105 106 107 108 109 110 111
Task2 Kicked Off
112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 
Task1 Done

Task3 Kicked Off
Task2 Started
201 202 203 204 205 206 207 208 209 210 211 212 213 301 214 215 216 217 218 302 219 220 221 222 223 
Task2 Done
306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 
Task3 Done

Main Done
```

Snippet-01 Explained

In this example, we implemented `Runnable` by implementing the `run()` method from `Runnable` interface.

To run Task2 which is implementing `Runnable` interface, we used this code. We are using the `Thread` constructor passing an instance of `Task2` .

```
Task2 task2 = new Task2();
Thread task2Thread = new Thread(task2);
task2Thread.start();
```

You can see from the output that all three tasks are running in parallel.

Summary

In this step, we:

- Explored another way to create threads, by implementing the `Runnable interface`
- Learned to run a thread created using `Runnable interface`

Step 03: The Thread Life-cycle

A Java Thread goes through a sequence of **states** during its lifetime. The term **life-cycle** is used to describe this fact, and clearly defines what specific state a thread could be in, at various points of time.

Let's consider the following example we explored recently, in ***Step 02***:

```
class Task1 extends Thread {
    public void run() {
        for(int i=101; i<=199; i++) {
            System.out.print(i + " ");
        }
    }
}

class Task2 implements Runnable {
    @Override
    public void run() {
        for(int i=201; i<=299; i++) {
            System.out.print(i + " ");
        }
    }
}

public class ThreadBasicsRunner {
    public static void main(String[] args) {
        Task1 task1 = new Task1();
        task1.start();
        Task2 task2 = new Task2();
        Thread task2Thread = new Thread(task2);
        task2Thread.start();
        for(int i=301; i<=399; i++) {
            System.out.print(i + " ");
        }
    }
}
```

Different states of a thread are:

- **NEW**: A thread is in this state as soon as it's been created, but its `start()` method hasn't yet been invoked.
    - For *Task1* : After the execution of `Task1 task1 = new Task1();`
    - For *Task2* : After the execution of `Task2 task2 = new Task2(); task2Thread = new Thread(task2);`
- **TERMINATED/DEAD**: When all the statements inside a thread's `run()` method have been been completed, that thread is said to have terminated.

A thread can be in any one of the remaining three states, after its `start()` method has been invoked.

- **RUNNING**: If the thread is currently running.
- **RUNNABLE**: If the thread is not currently running, but is ready to do so at any time.
- **BLOCKED/WAITING**: If the thread is not currently running on the processor, but is not ready to execute either. This may be the case if it's waiting for an external resource (such as a user's input) or another thread.

Summary

In this step, we:

- Discussed different states of a Thread with an example

Step 04: Thread Priorities

Java allows you to *request* the thread scheduler, to change the priority of a thread. The priority of any thread always lies in a fixed range - `MIN_PRIORITY = 1` to `MAX_PRIORITY = 10` . The default priority that's assigned to any thread, is `NORM_PRIORITY = 5` .

A request to change this priority is done by invoking the static `setPriority(int)` method, available in the `Thread class` . This request may or may not be honored in response, so be prepared for that!

Step 05: Communicating Threads

Any program where threads are not explicitly created is a single-threaded application. The thread that we refer to here is the *main thread*, executing the program's `main()` method.

Sometimes, threads might depend on one another.

Let consider the example from **Step 02**. We want to add a condition - *Task3* should execute only after *Task1* terminates.

ThreadBasicsRunner.java

```
class Task1 extends Thread {
    public void run() {
        System.out.println("Task1 Started ");
        for(int i=101; i<=199; i++) {
            System.out.print(i + " ");
        }
    }
}

class Task2 implements Runnable {
    @Override
    public void run() {
        System.out.println("Task2 Started ");
        for(int i=201; i<=299; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\nTask2 Done");
    }
}

public class ThreadBasicsRunner {
    public static void main(String[] args) {
        System.out.print("\nTask1 Kicked Off\n");
        Task1 task1 = new Task1();
        task1.start();
        System.out.print("\nTask2 Kicked Off\n");
        Task2 task2 = new Task2();
        Thread task2Thread = new Thread(task2);
        task2Thread.start();

        task1.join();

        System.out.print("\nTask3 Kicked Off\n");
        for(int i=301; i<=399; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\nTask3 Done");
```

```
            System.out.println("\nMain Done");
        }
    }
```

Snippet-5 Explained

`task1.join()` waits until task1 completes. So, the code after `task1.join()` will executed only on completion of `task1`.

If we want *Task3* to be executed only after both *Task1* and *Task2* are done, the code in `main()` needs to look as follows:

Snippet-6 : Task3 after Task1 and Task2

ThreadBasicsRunner.java

```
    public static void main(String[] args) {
        System.out.print("\nTask1 Kicked Off\n");
        Task1 task1 = new Task1();
        task1.start();

        System.out.print("\nTask2 Kicked Off\n");
        Task2 task2 = new Task2();
        Thread task2Thread = new Thread(task2);
        task2Thread.start();

        task1.join();
        task2Thread.join();

        System.out.print("\nTask3 Kicked Off\n");
        for(int i=301; i<=399; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\nTask3 Done");
        System.out.println("\nMain Done");

    }
```

Snippet-6 Explained

It is important to note that *Task1* and *Task2* are still independent sub-tasks. The thread scheduler is free to interleave their executions. However, *Task3* is kicked off only after both of them terminate.

Summary

In this step, we:

- Understood the need for thread communication
- Learned that Java provides mechanisms for threads to wait for each other
- Observed how the `join()` method can be used to sequence thread execution

Step 07: `synchronized` Methods, And `Thread` Utilities

When a thread gets tired, you can put it to bed. Heck, you can do it even when it's fresh and raring to go! It's under your control, remember?

The `Thread` class provides a couple of methods:

- `public static native void sleep(int millis)` : Calling this method will cause the thread in question, to go into a *blocked* / *waiting* state for **at least** `millis` milliseconds.

- `public static native void yield()` : Request thread scheduler to execute some other thread. The scheduler is free to ignore such requests.

Snippet-01 : Thread utilities

jshell> `Thread.sleep(1000)`

jshell> `Thread.sleep(10000)`

jshell>

Snippet-7 Explained

- `Thread.sleep(1000)` causes the `JShell` prompt to appear after a delay of *at least* 1 second. This delay is even more visible, when we execute `Thread.sleep(10000)` .

Step 08: Drawbacks of earlier approaches

We saw few of the methods for synchronization in the `Thread` class

- `start()`
- `join()`
- `sleep()`
- `wait()`

Above approaches have a few drawbacks:

- **No Fine-Grained Control**: Suppose, for instance , we want *Task3* to run after *any one* of *Task1* or *Task2* is done. How do we do it?
- **Difficult to maintain**: Imagine managing 5-10 threads with code written in earlier examples. It would become very difficult to maintain.
- **NO Sub-Task Return Mechanism**: With the `Thread class` or the `Runnable interface` , there is no way to get the result from a sub-task.

Step 09: Introducing `ExecutorService`

In order to address the serious limitations of the `Thread` API, a new `Executor Service` was added in **Java SE 5**.

The `ExecutorService` is a framework you can use to manage threads in your code, better. It has built-in ways to:

- Create and launch threads more intuitively
- Manage thread state and its life-cycle more easily
- Synchronize between threads with more control, and
- Handle groups of threads neatly

The `ExecutorService` provides utilities to achieve each one of these. Let's start with thread creation, which the next example takes care of.

Snippet-01 : Creating a Thread

ExecutorServiceRunner.java

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Task1 extends Thread {
    public void run() {
        System.out.println("Task1 Started ");
```

```
            for(int i=101; i<=199; i++) {
                System.out.print(i + " ");
            }
            System.out.println("\nTask1 Done");
        }
    }

    class Task2 implements Runnable {
        @Override
        public void run() {
            System.out.println("Task2 Started ");
            for(int i=201; i<=299; i++) {
                System.out.print(i + " ");
            }
            System.out.println("\nTask2 Done");
        }
    }


    public class ExecutorServiceRunner {
        public static void main(String[] args) {
            ExecutorService executorService = Executors.newSingleThreadExecutor();
            executorService.execute(new Task1());
            executorService.execute(new Thread(new Task2()));
        }
    }
```

Console Output

Task1 Started

101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199

Task1 Done

Task2 Started

201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299

Task2 Done

Snippet-01 Explained

`ExecutorService executorService = Executors.newSingleThreadExecutor()` creates a single threaded executor service. That's why the tasks ran serially, one after the other.

Snippet-02 : main runs in parallel

Let's update the main method to do more things:

```
    public class ExecutorServiceRunner {
        public static void main(String[] args) {
            ExecutorService executorService = Executors.newSingleThreadExecutor();
            executorService.execute(new Task1());
            executorService.execute(new Thread(new Task2()));
            System.out.print("\nTask3 Kicked Off\n");
            for(int i=301; i<=399; i++) {
                System.out.print(i + " ");
            }
            System.out.println("\nTask3 Done");
            System.out.println("\nMain Done");
            executorService.shutdown();
        }
    }
```

Console Output

```
Task1 Started
101
Task3 Kicked Off
102 301 103 302 104 303 105 304 106 305 107 306 108 109 110 111 112 307 113 114 115 116 117 118 119
Task1 Done
337 338 Task2 Started
339 201 202 203 204 205 206 207 340 341 208 209 210 211 212 213 214 215 216 217 218 219 220 342 221
Task3 Done

Main Done
292 293 294 295 296 297 298 299
Task2 Done
```

Snippet-02 Explained

The only order that we see in the resulting chaos is: *Task2* starts execution only after *Task1* is done.

Threads managed by `ExecutorService` run in parallel with `main` method.

Step 10: Executor - Customizing Number Of Threads

With the `ExecutorService`, it is possible to create a *pool* of threads.

The following examples will show you how you can create thread pools of varying kinds, and of course, of different sizes.

Snippet-03 : Executors for Concurrent threads

ExecutorServiceRunner.java

```
    public class ExecutorServiceRunner {
        public static void main(String[] args) {
            ExecutorService executorService = Executors.newFixedThreadPool(2);
            executorService.execute(new Task1());
            executorService.execute(new Thread(new Task2()));
            System.out.print("\nTask3 Kicked Off\n");

            for(int i=301; i<=399; i++) {
                System.out.print(i + " ");
            }
            System.out.println("\nTask3 Done");
            System.out.println("\nMain Done");
            executorService.shutdown();
        }
    }
```

Console Output

```
Task1 Started

Task3 Kicked Off
301 101 302 303 304 Task2 Started
305 306 102 307 103 201 104 308 105 202 106 309 107 203 108 310 109 110 204 111 311 112 205 113 312
Task3 Done

Main Done
184 185 279 280 281 282 283 186 187 284 285 286 188 189 287 288 289 190 191 290 192 291 193 292 194
Task2 Done
197 198 199
Task1 Done
```

Snippet-03 Explained

We created `ExecutorService` by using `Executors.newFixedThreadPool(2)` . So, 'ExecutorService` uses two parallel threads at a maximum.

- *Task1* and *Task2* execute concurrently as part of the `ExecutorService` , and
- The thread running `main()` executes concurrently with this thread pool, created by `ExecutorService` .

Snippet-04 : All-Executor Task Execution

Let's create a simple example to allow to play with 'ExecutorService'.

ExecutorServiceRunner.java

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Task extends Thread {
    private int number;

    public Task(int number) {
        this.number = number;
    }

    public void run() {
        System.out.println("Task " + number + " Started");
        for(int i=number*100; i<=number*100+99; i++) {
            System.out.print(i + " ");
        }
        System.out.println("\nTask " + number +" Done");
    }
}

public class ExecutorServiceRunner {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(2);
        executorService.execute(new Task(1));
        executorService.execute(new Task(2));
        executorService.execute(new Task(3));
        executorService.shutdown();
    }
}
```

Snippet-04 Explained

`Executors.newFixedThreadPool(2)` - Two threads in parallel. The thread `new Task(3)` is executed only after any one of `new Task(1)` and `new Task(2)` have completed their execution.

Snippet-04 : Larger Thread Pool Size

```
public class ExecutorServiceRunner {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(3);
        executorService.execute(new Task(1));
        executorService.execute(new Task(2));
        executorService.execute(new Task(3));
        executorService.execute(new Task(4));
        executorService.execute(new Task(5));
        executorService.execute(new Task(6));
        executorService.execute(new Task(7));
        executorService.shutdown();
    }
}
```

Snippet-04 Explained

We made the pool larger to 3:

- Initially Tasks `1` , `2` , `3` are added to the `ExecutorService` and are started.
- As soon as any one of them is terminated, another task from the thread pool is picked up for execution, and so on. A classic case of musical chairs, with regard to the slots available in the `ExecutorService` thread pool, is played out here!

Summary

In this step, we:

- Explored how one can create pools of threads of the same kind, using the `ExecutorService`
- Noted how one could specify the size of a thread pool

Step 11: `ExecutorService` : Returning Values From Tasks

Snippet-01: Returning a Future Object

So far, we have only seen sub-tasks that are largely independent, and which don't return any result to the main program that launched them.

To be able to return a value from a Thread, Java provides a `Callable<T>` interface.

The next example tells you how to implement `Callable<T>` , and use it with `ExecutorService` .

ExecutorServiceRunner.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Callable;

class CallableTask implements Callable<String> {
    private String name;

    public CallableTask(String name) {
        this.name = name;
    }

    @Override
    public String call() throws Exception {
        Thread.sleep(1000);
        return "Hello " + name;
    }
}
```

```
public class CallableRunner {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newFixedThreadPool(1);
        Future<String> welcomeFuture = executorService.submit(new CallableTask("in28Minutes"));
        System.out.println("CallableTask in28Minutes Submitted");
        String welcomeMessage = welcomeFuture.get();
        System.out.println(welcomeMessage);
        executorService.shutdown();
    }
}
```

Console Output

CallableTask in28Minutes Submitted

Hello in28Minutes

Snippet-01 Explained

- `class CallableTask implements Callable<String>` - Implement `Callable`. Return type is `String`
- `public String call() throws Exception {` - Implement `call` method and return a `String` value back.
- `executorService.submit(new CallableTask(String))` adds a callable task to its thread pool. This puts the task thread in a **RUNNABLE** state. Subsequently, the program goes ahead to invoke its `call()` method.
- `welcomeFuture.get()` - Ensures that the `main` thread waits for the result of the operation.

Summary

In this step, we:

- Understood the need for a mechanism, to create sub-tasks that return values
- Discovered that Java has a `Callable<T>` interface to do exactly this
- Saw that the `ExecutorService` is capable of managing `Callable` threads

Step 11: Executors - Waiting For Many Callable Tasks To Complete

`ExecutorService` framework also allows you to create a pool of `Callable` threads. Not only that, you can collect their return values together.

Snippet-01 : Waiting for Multiple `Callable` Threads

MultipleCallableRunner.java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Callable;

class CallableTask implements Callable<String> {
    private String name;

    public CallableTask(String name) {
        this.name = name;
    }

    @Override
    public String call() throws Exception {
        Thread.sleep(1000);
        return "Hello " + name;
    }
}
```

```
public class MultipleCallableRunner {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newFixedThreadPool(1);
        List<CallableTask> tasks = List.of(new CallableTask("in28Minutes"),
                                            new CallableTask("Ranga"),
                                            new CallableTask("Adam"));
        List<Future<String>> welcomeAll = executorService.invokeAll(tasks);
        for(Future<String> welcomeFuture : welcomeAll) {
            System.out.println(welcomeFuture.get());
        }
        executorService.shutdown();
    }
}
```

Console Output

Hello in28Minutes

Hello Ranga

Hello Adam

Snippet-01 Explained

The `invokeAll()` method of `ExecutorService` allows for a list of `Callable` tasks to be launched in the thread pool. Also, a `List` of `Future` objects can be used to hold return values, one for each such `Callable` thread.

The list of return values can be accessed only after **all** the threads are done, and have returned their results.

This can be verified from the console output. all the planned welcome messages are printed in one go, but only after a wait of at least `3000` milliseconds has been completed.

Let's now see what scenario would pan out with a larger thread pool size.

Snippet-02 : List of Callable tasks with larger thread pool

```
public class MultipleCallableRunner {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newFixedThreadPool(3);
        List<CallableTask> tasks = List.of(new CallableTask("in28Minutes"),
                                            new CallableTask("Ranga"),
                                            new CallableTask("Adam"));
        List<Future<String>> welcomeAll = executorService.invokeAll(tasks);
        for(Future<String> welcomeFuture : welcomeAll) {
            System.out.println(welcomeFuture.get());
        }
        executorService.shutdown();
    }
}
```

Console Output

Hello in28Minutes

Hello Ranga

Hello Adam

Snippet-02 Explained

The welcome messages all get printed in a batch again, but their collective wait gets shorter. This is because:

- The thread pool size is now `3`, not `2` as earlier. This means all the three tasks can be put in the **RUNNABLE** state at once.
- Then, they go into their **BLOCKED** state also almost simultaneously, which means their collective wait time is much less than `3000` milliseconds. That's one advantage of a larger thread pool!

Summary

In this step, we:

- Learned that it's possible to collect the return values of a pool of `Callable` threads, at one go.
- This is done using the `invokeAll()` method for their launch, and specifying a `List` of `Future` objects to hold these results
- Changing the thread pool size for such scenarios can change response time dramatically

Step 12: Executor - Wait Only For The Fastest Task

Let's look at how you can wait for any of the three tasks to complete.

Snippet-01 : Wait only for fastest

```
public class MultipleAnyCallableRunner {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newFixedThreadPool(3);
        List<CallableTask> tasks = List.of(new CallableTask("in28Minutes"),
                                            new CallableTask("Ranga"),
                                            new CallableTask("Adam"));
        String welcomeMessage = executorService.invokeAny(tasks);
        System.out.println(welcomeMessage);
        executorService.shutdown();
    }
}
```

Console Output

Hello Ranga

Console Output

Hello in28Minutes

Console Output

Hello Ranga

Console Output

Hello Adam

Snippet-01 Explained

The method `invokeAny()` returns when the first of the sub-tasks is done. Also, the returned value is not a `Future` object. It's the return value of the `call()` method.

We can see that over different executions, the order of console output changes. This is because:

- All three tasks are created together, in a thread pool of size `3`.
- Therefore, these are independent threads, going into their **RUNNABLE** states almost at once.

Summary

In this step, we:

- Learned that `ExecutorService` has a way to return the first result, from a poll of `Callable` threads

Introduction To Exception handling

Recommended Exception handling Videos

- https://www.youtube.com/watch?v=34ttwuxHtAE

There are two kinds of errors a programmer faces:

- **Compile-time** Errors: Flagged by the compiler when it detects syntax or semantic errors.
- **Run-time** Errors: Detected by the run-time environment when executing code

Example runtime errors include:

- Running out of *heap-memory* for objects, or space for the method *call stack*
- Dividing a number by `0`
- Trying to *read* from an *unopened* file

Exceptions are *unexpected* conditions; their occurrence does not make a programmer bad (or good, for that matter). It is a programmer's responsibility to be *aware* of potential exceptional conditions in her program, and use a mechanism to *handle* them effectively.

Handling an exception generally involves two important aims:

1. Provide a useful message to the end user.
2. Log enough information to help a programmer identify root cause.

Step 01: Introducing Exceptions

In the previous step, we gave you a few instances of exceptions, such as your program running out of memory, or your code trying to divide a number by `0` .

Want to see a live example of the Java run-time throwing an exception? The next example will satisfy your thirst.

Snippet-1 : Exception Condition

ExceptionHandlingRunner.java

```
    package com.in28minutes.exceptionhandling;

    public class ExceptionHandlingRunner {
        public static void main(String[] args) {
            callMethod();
            System.out.println("callMethod Done");
        }

        static void callMethod() {
            String str = null;
            int len = str.length();
            System.out.println("String Length Done");
        }
    }
```

Console Output

java.lang.NullPointerException

Exception in thread "main" java.lang.NullPointerException

at com.in28minutes.exceptionhandling.ExceptionHandlingRunner.callMethod (ExceptionHandlingRunner.java:8)

at com.in28minutes.exceptionhandling.ExceptionHandlingRunner.main (ExceptionHandlingRunner.java:4)

Snippet-01 Explained

A `java.lang.NullPointerException` is **thrown** by the Java run-time when we called `length()` on the `null` reference.

If an exception is not handled in the entire call chain, including `main` , the exception is thrown out and the program terminates. `System.out.println()` statements after the exception are never executed.

The runtime prints the **call stack trace** onto the console.

For instance, consider this simple `class Test` :

```
public class Test {
    public static void main(String[] args) {
        callOne();
    }

    public static void callOne() {
        callTwo();
    }

    public static void callTwo() {
        callThree();
    }

    public static void callThree() {
        String name;
        System.out.println("This name is %d characters long", name.length());
    }
}
```

However, the code name.length() used in `callThree()` caused a `NullPointerException` to be thrown. However, this is not handled, and the console coughs up the following display:

Exception in thread "main" java.lang.NullPointerException

at Test.callThree(Test.java:13)

at Test.callTwo(Test.java:9)

at Test.callOne(Test.java:6)

at Test.main(Test.java:3)

This is nothing but a call trace of the stack, *frozen in time*.

Summary

In this step, we:

- Saw a live example of an exception being thrown
- Understood how a call trace on the stack appears when a exception occurs
- Reinforced this understanding with another example

Step 02: Handling An Exception

In Java, exception handling is achieved through a **`try - catch`** *block*.

Snippet-01 : Handling An Exception

ExceptionHandlingRunner.java

```
package com.in28minutes.exceptionhandling;

public class ExceptionHandlingRunner {
    public static void main(String[] args) {
        method1();
        System.out.println("main() Done");
    }

    static void method1() {
        method2();
        System.out.println("method1() done");
    }

    static void method2() {
        try {
            String str = null;
            int len = str.length();
            System.out.println("method2() Done");
        } catch (Exception ex) {
        }
    }
}
```

Console Output

method1() Done

main() Done

Snippet-01 Explained

We have handled an exception here with a `try - catch` block. Its syntax resembles this:

```
try {
    //< program-logic code block >
} catch (Exception e) {
    //< exception-handling code block >
}
```

The exception (the `NullPointerException`) still occurs after adding this block, but now it's actually **caught**. Although we did nothing with the caught exception, we did avoid a sudden end of the program.

The statements following `int len = str.length()` in `method2` are not executed.

However, all of `method1()` 's code was run (with the `"method1 Done"` message getting printed). `main()` method also completed execution successfully.

The program thus terminated gracefully. `method1()` and `main()` are both unaware of the `NullPointerException` occurring within `method2()` .

Snippet-02: Print Debug Information

Let's add `ex.printStackTrace();` .

ExceptionHandlingRunner.java

```
package com.in28minutes.exceptionhandling;

public class ExceptionHandlingRunner {
    public static void main(String[] args) {
        method1();
        System.out.println("main() Done");
    }

    static void method1() {
        method2();
        System.out.println("method1() done");
    }

    static void method2() {
        try {
            String str = null;
            int len = str.length();
            System.out.println("method2() Done");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Console Output

java.lang.NullPointerException

Exception in thread "main" java.lang.NullPointerException

at com.in28minutes.exceptionhandling.ExceptionHandlingRunner.method2 (ExceptionHandlingRunner.java:14)

at com.in28minutes.exceptionhandling.ExceptionHandlingRunner.method1 (ExceptionHandlingRunner.java:8)

at com.in28minutes.exceptionhandling.ExceptionHandlingRunner.main (ExceptionHandlingRunner.java:4)

method1() Done

main() Done

`printStackTrace()` is provided by the `Exception` `class`, which every exception inherits from. This method prints the frozen call trace of the program when the exception occurred, but without terminating the program. The code next continues to run.

The stack trace provides useful information to you, the programmer,to debug the exception scenario.

Summary

In this step, we:

- Were introduced to Java's basic mechanism to handle exceptions, the `try` - `catch` block
- Saw how a program runs and ends gracefully, when an exception is handled
- Observed how the `printStackTrace` method gives debug information to the programmer

Step 03: The `Exception` Hierarchy

The code in the `try` - `catch` block above does not work by black magic. From the call trace, it's clear that this program encounters a `NullPointerException` in `method1()`. What's surprising, is that it was caught by a `catch` clause meant for an `Exception`! The reason this worked is because `NullPointerException` **is-a** `Exception`.

You heard right! Java has a hierarchy of exception types, rooted at `class` `Exception`. For instance,

- `NullPointerException` **is-a** `RuntimeException` , and
- `RuntimeException` **is-a** `Exception` .
- So effectively, `NullPointerException` **is-a** `Exception` !

Different branches of this inheritance-tree actually denote different exception categories. We'll dwell on this topic a little later.

Snippet-01 : Catching NullPointerException

Let's add an additional catch for `NullPointerException` in method2.

ExceptionHandlingRunner.java

```
package com.in28minutes.exceptionhandling;

public class ExceptionHandlingRunner {
    public static void main(String[] args) {
        method1();
        System.out.println("main() Done");
    }

    static void method1() {
        method2();
        System.out.println("method1() done");
    }

    static void method2() {
        try {
            String str = null;
            int len = str.length();
            System.out.println("method2() Done");
        } catch (NullPointerException e) {
            System.out.println("NullPointerException");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Console Output

NullPointerException

method1() Done

main() Done

Snippet-01 Explained

*Among all the `catch` clauses following the `try` , **one and only one** of them, may get executed. The **first** `catch` clause to **match, in serial order** after the `try` , always gets executed. If **none** match, the exception is **not handled**.*

We placed an additional `catch` block within `method2()` , to handle `NullPointerException` . Typically, the most specific exception class is matched. Hence the catch block for `NullPointerException` matches.

You need to order the `catch` blocks after a `try` , from more-specific to less-specific matches.

Snippet-02 : Catching another exception

```
package com.in28minutes.exceptionhandling;
```

```
public class ExceptionHandlingRunner {
    public static void main(String[] args) {
        method1();
        System.out.println("main() Done");
    }

    static void method1() {
        method2();
        System.out.println("method1() done");
    }

    static void method2() {
        try {
            int[] numbers = {1, 2};
            int num = numbers[3];
            System.out.println("method2() Done");
        } catch (NullPointerException e) {
            System.out.println("NullPointerException");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Console Output

ArrayIndexOutOfBoundsException : 3

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

at com.in28minutes.exceptionhandling.ExceptionHandlingRunner.method2 (ExceptionHandlingRunner.java:14)

at com.in28minutes.exceptionhandling.ExceptionHandlingRunner.method1 (ExceptionHandlingRunner.java:8)

at com.in28minutes.exceptionhandling.ExceptionHandlingRunner.main (ExceptionHandlingRunner.java:4)

method1() Done

main() Done

Snippet-02 Explained

`ArrayIndexOutOfBoundsException` **is-a** `IndexOutOfBoundsException` , which **is-a** `RuntimeException` , which in turn **is-a** `Exception` .

`ArrayIndexOutOfBoundsException` is not a sub class of `NullPointerException` . Hence, it does not match with the first catch block. `ArrayIndexOutOfBoundsException` is a sub class of `Exception` . Hence, that `catch` block matched, and the statement `ex.printStackTrace();` within it ran.

If we omit the handler for `Exception` , the `ArrayIndexOutOfBoundsException` is not caught by this `try - catch` block. Since it is not handled in `method1()` , or even later in `main()` , our program would have to stop suddenly.

Summary

In this step, we:

- Learned that there is an exception hierarchy in Java, rooted at `Exception`
- Looked at an example that could cause multiple exceptions to occur
- Observed how a handler for `Exception` could match any exception

Step 04: The Need For `finally`

When an exception occurs, the programmer's world can turn upside-down in a matter of moments. If not handled, the program terminates all of a sudden, with no light at the end of the tunnel.

Snippet-01 : Unreleased Resources

FinallyRunner.java

```
package com.in28minutes.exceptionhandling;
import java.util.Scanner;

public class FinallyRunner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // ... Program logic, probably using scanner input
        int num = numbers[5];
        System.out.println("Before scanner close");
        scanner.close();
    }
}
```

Console Output

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

at com.in28minutes.exceptionhandling.FinallyRunner.main (FinallyRunner.java:8)

Snippet-01 Explained

This example makes use of `Scanner` object to read from console. Ideally a `Scanner` object should be closed using `scanner.close();` .

However, in our example, it is not called because a line before it threw an exception. (`int num = numbers[5];` tries to access the 5th element of a 4-element array).

What this means, is a system resource that has been acquired, is never released.

It's important to ensure that any acquired resource is always released; whether on normal or abrupt termination. Let's see how you do this while handling an exception.

Snippet-02 : Releasing Resources

FinallyRunner.java

```
package com.in28minutes.exceptionhandling;
import java.util.Scanner;

public class FinallyRunner {
    public static void main(String[] args) {
        Scanner scanner = null;
        try {
            scanner = new Scanner(System.in);
            // ... Program logic, probably using scanner input
            int[] numbers = {1, 2, 3, 4};
            int num = numbers[5];
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if(scanner != null) {
                System.out.println("Before scanner close");
                scanner.close();
            }
        }
```

```
            System.out.println("Before exiting main");
        }
    }
```

Console Output

ArrayIndexOutOfBoundsException

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

at com.in28minutes.exceptionhandling.FinallyRunner.main (FinallyRunner.java:10)

Before scanner close

Before exiting main

Snippet-02 Explained

Code in `finally` is almost always executed - even when there are exceptions.

We added a null check on `scanner` since `scanner = new Scanner(System.in);` could also result in an exception.

```
    } finally {
        if(scanner != null) {
            System.out.println("Before scanner close");
            scanner.close();
        }
    }
```

Summary

In this step, we:

- Observed how exceptional conditions could result in resource leaks
- Learned about the `finally` clause in a `try` - `catch` block

Step 05: Programming Puzzles - PP-01

Puzzle-01

- Would the `finally` clause be executed if
  - The statement `//str = "Hello";` remains as-is
  - The statement `//str = "Hello";` has its comments removed?

```
    public static void method3() {
        Connection connection = new Connection();
        connection.open();
        try {
            String str = null;
            //str = "Hello";
            str.toString();
            return;
        } catch(Exception e) {
        } finally {
            connection.close();
        }
    }
```

- ***Answer***

    - Yes
    - Yes

Puzzle-02

- When will code in a `finally` clause not get executed?
- ***Answer***
    - In case of statements within the same `finally` clause, preceding this code, throwing an exception
    - In case of a JVM crash. This can be simulated in some scenarios by calling `System.exit` with an appropriate `int` argument, within an appropriate `catch` clause of the same `try` - `catch` - `finally` clause.

Puzzle-03

- Will the following code, a `try` - `finally` without a `catch` ?

```
public static void method4() {
    Connection connection = new Connection();
    connection.open();
    try {
        String str = null;
        //str = "Hello";
        str.toString();
        return;
    } finally {
        connection.close();
    }
}
```

- ***Answer*** : Yes

Puzzle-04

- Will the following code, a `try` without a `catch` or a `finally` ?

```
public static void method5() {
    Connection connection = new Connection();
    connection.open();
    try {
        String str = null;
        //str = "Hello";
        str.toString();
        return;
    }
}
```

- ***Answer*** : No

Step 06: Handling Exceptions: Do We Have A Choice?

Sometimes, in Java you are forced to handle exceptions.

Snippet-02: Checked Exceptions - v1

CheckedExceptionRunner.java

```
package com.in28minutes.exceptionhandling;

public class CheckedExceptionRunner {
    public static void main(String[] args) {
        Thread.sleep(2000);
    }
}
```

Snippet-02 Explained

This program will not compile!

The reason we get flagged by a compiler error, lies in the signature of the `sleep()` method. Here is its definition within the `Thread` `class` :

```
public static native void sleep(long millis) throws InterruptedException {
    //...
}
```

This declaration is a bit different from what you normally expect for a method, isn't it! It contains a **`throws` specification**.

When a method `throws` an Exception, the calling method should:

- Handle it using `try` - `catch` block
- Or declare `throws` in its signature

Let's use `try` - `catch` block to start off.

Snippet-03: Checked Exceptions - v2

CheckedExceptionRunner.java

```
package com.in28minutes.exceptionhandling;

public class CheckedExceptionRunner {
    public static void main(String[] args) {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Snippet-03 Explained

- `main()` , which is the caller of `sleep()` , chooses the option of handling `InterruptedException` with a `try` - `catch` block.

The `throws` keyword

`throws` is used to declare that a method might throw exceptions. This involves a `throws` keyword, followed by a list of exception types.

Snippet-04 : Method with risky code #1

```
package com.in28minutes.exceptionhandling;

public class CheckedExceptionRunner {
    public static void main(String[] args) {
        try {
            riskyMethod();
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void riskyMethod() throws InterruptedException {
        Thread.sleep(5000);
    }
}
```

Snippet-04 Explained

Here, we have removed the `try` - `catch` block within `riskyMethod()` , because we want to follow another way of managing the exception. As an alternative, we added a `throws` specification to `riskyMethod()` to make the code compile.

We made `main` method handle the exception.

Summary

In this step, we:

- Discovered that certain exceptions in Java do not force you to handle them
- Learned that all the rest must be managed/handled
- Observed that there are two ways to manage those "Checked" exceptions:
  - Handling with a `try` - `catch` block
  - Using a `throws` specification

Step 08: The Java Exception Hierarchy

Right at the root (top, we mean), the Java exception hierarchy looks like this:

```
class Error extends Throwable{}
class Exception extends Throwable{}
class InterruptedException extends Exception{}
class RuntimeException extends Exception{}
class NullPointerException extends RuntimeException{}
...
```

Once an `Error` occurs, there is nothing a programmer could do. Examples include:

- The JVM running out of heap memory space

An `Exception` can be handled. There are two types of Exceptions.

- `RuntimeException` and its sub-classes. These come under the category of **unchecked exceptions**. Another example we've seen is `NullPointerException` , which inherits from `RuntimeException` .
- All other sub-classes of `Exception` , excluding the sub-tree rooted at `RuntimeException` , are called **checked exceptions**. An instance we've encountered is `InterruptedException` .

If a method throws a **checked exception** is called, then either:

- The method call must be enclosed in a `try` - `catch` block for proper handling, or
- The caller must throw this exception out, to its own caller. Its signature must also be enhanced using a `throws` specification.

If an **unchecked exception** is involved, then:

- You have the options of handling with `try` - `catch` block.
- It is not mandatory to handle it.

A **checked** exception **must** be handled, whereas as **unchecked** exception **may or may not** be handled.

Summary

In this step, we:

- Discovered that within the Java exception hierarchy, there are two categories:
  - Checked exceptions
  - Unchecked exceptions
- There are different strategies to manage these two categories

Step 09: Throwing an Exception

So far, we have seen how to handle an exception, that is thrown by a built-in Java method. Now, let's explore how we can alert a user about exceptional conditions in our own code, by **throwing** exceptions.

Snippet-01 : Throwing An Exception

ThrowingExceptionRunner.java

```java
package com.in28minutes.exceptionhandling;

class Amounts {
    private String currency;
    private int amount;

    public Amounts(String currency, int amount) {
        super();
        this.currency = currency;
        this.amount = amount;
    }

    public void add(Amount that) {
        if(!this.currency.equals(that.currency)) {
            throw new RuntimeException("Currencies Don't Match");
        }
        this.amount += that.amount;
    }

    public Sring toString() {
        return amount + " " + currency;
    }
}

public class ThrowingExceptionRunner {
    public static void main(String[] args) {
        Amount amount1 = new Amount("USD", 10);
        //Amount amount2 = new Amount("USD", 20);
        Amount amount2 = new Amount("EUR", 20);
        amount1.add(amount2);
        System.out.println(amount1);
    }
}
```

Console Output

Exception in thread "main" java.lang.RuntimeException:Currencies Don't Match

at com.in28minutes.exceptionhandling.ThrowingExceptionRunner.main (ThrowingExceptionRunner.java:26)

Snippet-01 Explained

Since adding `10 USD` and `20 EUR` does not make sense in the real world, it's important to tell the user that `add()` won't work for different currencies. The most direct way is to throw an exception, which the code `throw new RuntimeException("Currencies Don't Match");` does.

This thrown exception object can be handled inside `main()` . By calling `printStackTrace()` on the caught exception reference, you get debug information like before.

Snippet-02 : Throwing a Checked Exception

`Exception` is a Checked Exception. If a method throws an instance of `Exception` class, it needs to declare it - `public void add(Amount that) throws Exception {` .

ThrowingExceptionRunner.java

```java
package com.in28minutes.exceptionhandling;

class Amounts {
    private String currency;
    private int amount;

    public Amounts(String currency, int amount) {
        super();
        this.currency = currency;
        this.amount = amount;
    }

    public void add(Amount that) throws Exception {
        if(!this.currency.equals(that.currency)) {
            throw new Exception("Currencies Don't Match : " + this.currency + " &  that.currency
        }
            this.amount += that.amount;
    }

    public String toString() {
        return amount + " " + currency;
    }
}

public class ThrowingExceptionRunner {
    public static void main(String[] args) {
        Amount amount1 = new Amount("USD", 10);
        //Amount amount2 = new Amount("USD", 20);
        Amount amount2 = new Amount("EUR", 20);
    try {
            amount1.add(amount2);
            System.out.println(amount1);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Console Output

java.lang.RuntimeException:Currencies Don't Match : USD & EUR

Exception in thread "main" java.lang.RuntimeException:Currencies Don't Match

at com.in28minutes.exceptionhandling.ThrowingExceptionRunner.main (ThrowingExceptionRunner.java:26)

`Exception` is not a `RuntimeException` or one of its sub-classes, it is a checked exception. So, it needs to be declared when it is thrown - `public void add(Amount that) throws Exception`.

Summary

In this step, we:

- Learned that it is possible to throw an exception from code inside any method we write
- When a method throws checked exception, it should declare it.

Step 10: Throwing A Custom Exception

It is also possible for you to throw a custom exception. You can do this by defining your own exception `class`, only that it must inherit from one of the built-in exception classes. Note that:

- If you sub-class a checked exception, your exception also becomes checked.
- If you sub-class an unchecked exception, your exception would be unchecked.

Snippet-01 : Throw a custom exception

ThrowingExceptionRunner.java

```
package com.in28minutes.exceptionhandling;

class CurrenciesDoNotMatchException extends Exception {
    public CurrenciesDoNotMatchException(String msg) {
        super(msg);
    }
}

class Amounts {
    private String currency;
    private int amount;

    public Amounts(String currency, int amount) {
        super();
        this.currency = currency;
        this.amount = amount;
    }

    public void add(Amount that) throws Exception {
        if(!this.currency.equals(that.currency)) {
            throw new CurrenciesDoNotMatchException("Currencies Don't Match : " + this.currency -
        }
        this.amount += that.amount;
    }

    public String toString() {
        return amount + " " + currency;
    }
}

public class ThrowingExceptionRunner {
    public static void main(String[] args) {
        Amount amount1 = new Amount("USD", 10);
        //Amount amount2 = new Amount("USD", 20);
        Amount amount2 = new Amount("EUR", 20);
```

```
            try {
                amount1.add(amount2);
                System.out.println(amount1);
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Console Output

com.in28minutes.exceptionhandling.CurrenciesDoNotMatchException : Currencies Don't Match : USD & EUR

Exception in thread "main" com.in28minutes.exceptionhandling.CurrenciesDoNotMatchException : Currencies Don't Match : USD & EUR

at com.in28minutes.exceptionhandling.ThrowingExceptionRunner.main (ThrowingExceptionRunner.java:26)

Snippet-01 Explained

The class `CurrenciesDoNotMatchException` clearly is a checked exception. Hence, rules that apply for throwing and handling checked applications, apply to it as well.

If instead, `CurrenciesDoNotMatchException` were to sub-class `RuntimeException` , it would be an unchecked exception. Adding the `throws` specification to the `add()` definition would not be needed. Also, no method in the call sequence of `add()` is required to handle it.

Summary

In this step, we:

- Discovered that Java allows you to define your own custom exception classes
- Whether a custom exception is checked or unchecked, depends on which exception it sub-classes.
- Saw an example of how to raise and handle a custom exception

Step 11: Introducing `try` -With-Resources

`try` -with-resources makes managing resource in a `try` - `catch` - `finally` block. Let's look at an example.

Snippet-01: try-with-resources

```
    package com.in28minutes.exceptionhandling;
    import java.util.Scanner;

    public class TryWithResourcesRunner {
        public static void main(String[] args) {
            try (Scanner scanner = new Scanner(System.in)) {
                // ... Program logic, probably using scanner input
                int[] numbers = {1, 2, 3, 4};
                int num = numbers[5];
            }
            //scanner.close();
        }
    }
```

Snippet-01 Explained

The `try` -with-resources version was introduced in Java SE 7. It encloses the resource to be managed within parentheses, along with the `try` keyword.

In this case, `Scanner` is compatible with such resource management, because it's defined to implement the `Closeable interface`. This interface in turn, is a sub-class of the `abstract class AutoCloseable`.

`Closeable extends AutoCloseable{};`

`Scanner implements Closeable{};`

For `try`-with-resources, a `catch` and/or a `finally` clause are not mandatory.

Also, the call to `scanner.close` is no longer required.

Summary

In this step, we:

- Discovered a veriant of the `try`-`catch`-`finally` block, called `try`-with-resources
- Saw that it can be used to manage resource cleanly, provided the resource implements the `AutoCloseable interface`.

Step 12: Programming Puzzle Set PP_02

Puzzle-01

- Does the following program handle the exception thrown?

```
    try {
            AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("RUPEE", 5);
            String str = null;
            str.toString();
        } catch(CurrenciesDoNotMatchException ex) {
            ex.printStackTrace();
        }
```

- ***Answer : No***

Puzzle-01 Explained

The exception thrown is a `NullPointerException`, whereas the one we are trying to catch is a `CurrenciesDoNotMatchException`.

Puzzle-02

- Does the following code compile?

```
    try {
            AmountAdder.addAmounts(new Amount("RUPEE", 5), new Amount("RUPEE", 5);
            String str = null;
            str.toString();
        } catch(Exception e) {
            e.printStackTrace();
        } catch(CurrenciesDoNotMatchException ex) {
            ex.printStackTrace();
        }
```

Answer : No

Puzzle-02 explained

The order of `catch` clauses for exceptions needs to be from less specific to more specific. `CurrenciesDoNotMatchException` is a sub-class of `Exception`. Hence, error.

Puzzle-03

- Does the following code compile?

```
try {

} catch (IOException | SQLException ex) {
    ex.printStackTrace();
}
```

- *Answer : Yes*

Puzzle-03 Explained

This feature was added in Java SE 7.

File Operations

We would be aware that any computer has a hard disk, on which information is stored. This information is stored in units called **files**. For ease of access, file are grouped together and organized into **directories**. The operating system has a sub-system called the **file-system**, which has the responsibility of interfacing system and user programs, with files and directories on disk.

The Java Runtime System also communicates with the native file-system, and makes use of its services to provide a file programming interface to Java programmers.

In this section, we will explore a few basic ways in which programmers can interact with the native file-system, through the platform independent Java file API.

Listing Directory Contents

When we develop a Java software project in the Eclipse IDE environment, the root folder of the project has several interesting file and sub-folders contained within it. From now on, we use the terms "folder" and "directory" interchangeably, as they have equivalent meanings.

Let's write a simple Java program to list the contents of one such project folder.

Snippet-1 : Listing Directory Contents

The `java.nio.file` system package has a bunch of utility `class` es and `interface` s to help us navigate the native file system.

DirectoryScanRunner.java

```
package com.in28minutes.files;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;

public class DirectoryScanRunner {
    public static void main(String[] args) throws IOException {
        Files.list((Paths.get("."))).forEach(System.out::println);
    }
}
```

Console Output

./.classpath

./.project

_./.DS_Store_

./bin

./resources

./src

Snippet-1 Explained

A `Path class` is the entity used to denote a **pathname** in Java NIO Library. NIO stands for "New I/O", which was introduced in Java SE, replacing the earlier, very awkward File I/O Library. It is no longer new though, but that's another issue!

" **.** " denotes the current directory in the file-system.

The `Paths.get()` method returns the path-name of the specified directory in a format that the `Files.get()` method understands.

The `Files.get()` method does a **lazy traversal** of the directory it is provided with, in the sense that:

- It lists regular files it encounters.
- When faced with directory files, it merely lists them, without recursively traversing them.

The contents of the root directory are listed as path-names, with each path-name being relative to the root directory.

Snippet-2 : Recursive Directory Traversal

We can specify level 2 in `Files.walk(currentDirectory, 2)` . So, folders until level 2 are scanned.

DirectoryScanRunner.java

```
package com.in28minutes.files;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class DirectoryScanRunner {
    public static void main(String[] args) throws IOException {
        Path currentDirectory = Paths.get(".");
        //Files.list(currentDirectory).forEach(System.out::println);
        Files.walk(currentDirectory, 2).forEach(System.out::println);
    }
}
```

Console Output

./.classpath

./.project

_./.DS_Store_

./bin

./bin/files

./resources

./src

./src/files

Snippet-3 : Level-4 Traversal

DirectoryScanRunner.java

```
package com.in28minutes.files;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.io.IOException;

public class DirectoryScanRunner {
    public static void main(String[] args) throws IOException {
        Path currentDirectory = Paths.get(".");
        Files.walk(currentDirectory, 4).forEach(System.out::println);
    }
}
```

Console Output

./.classpath

./.project

_./.DS_Store_

./bin

./bin/files

./bin/files/DirectoryScanRunner.class

./resources

./src

./src/files

./src/files/DirectoryScanRunner.java

Snippet-4 : Only list .java files during traversal

We use a predicate `Files.walk(currentDirectory, 4).filter(predicate)` to filter only Java files.

DirectoryScanRunner.java

```
package com.in28minutes.files;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.function.Predicate;
import java.io.IOException;

public class DirectoryScanRunner {
    public static void main(String[] args) throws IOException {
        Path currentDirectory = Paths.get(".");
        Predicate<? super Path> predicate = path -> String.valueOf(Path).contains(".java");

        //Files.walk(currentDirectory, 4).forEach(System.out::println);
        Files.walk(currentDirectory, 4).filter(predicate)
                                       .forEach(System.out::println);
    }
}
```

Console Output

./src/files/DirectoryScanRunner.java

Snippet-5 : Filtered Traversal with find()

We can use a matcher - `Files.find(currentDirectory, 4, matcher)` which is configured to check the path attribute for .java extension - `(path, attributes) -> String.valueOf(path).contains(".java")`

DirectoryScanRunner.java

```
    package com.in28minutes.files;
    import java.nio.file.Files;
    import java.nio.file.Path;
    import java.nio.file.Paths;
    import java.util.function.Predicate;
    import java.nio.file.attribute.BasicFileAttributes;
    import java.util.function.BiPredicate;
    import java.io.IOException;

    public class DirectoryScanRunner {
        public static void main(String[] args) throws IOException {
            Path currentDirectory = Paths.get(".");
            //Predicate<? super Path> predicate = path -> String.valueOf(path).contains(".java");
            //Files.walk(currentDirectory, 4).filter(predicate).forEach(System.out::println);
            BiPredicate<Path, BasicFileAttributes> matcher =
            (path, attributes) -> String.valueOf(path).contains(".java");
            Files.find(currentDirectory, 4, matcher);
        }
    }
```

Console Output

./src/files/DirectoryScanRunner.java

Snippet-6 : Filtering directories

DirectoryScanRunner.java

```
    package com.in28minutes.files;
    import java.nio.file.Files;
    import java.nio.file.Path;
    import java.nio.file.Paths;
    import java.util.function.Predicate;
    import java.nio.file.attribute.BasicFileAttributes;
    import java.util.function.BiPredicate;
    import java.io.IOException;

    public class DirectoryScanRunner {
        public static void main(String[] args) throws IOException {
            Path currentDirectory = Paths.get(".");
            //BiPredicate<Path, BasicFileAttributes> matcher =
            //(path, attributes) -> String.valueOf(path).contains(".java");
            BiPredicate<Path, BasicFileAttributes> directoryMatcher =
            (path, attributes) -> attributes.isDirectory();
            Files.find(currentDirectory, 4, directoryMatcher);
        }
    }
```

Console Output

./bin

./bin/files

./resources/

./src/

./src/files

We are making use of a matcher checking `attributes.isDirectory()` .

Snippet-7 : Reading a File

./resources/data.txt

```
123.122

asdfghjkl

Apple

Bat

Cat
```

FileReadRunner.java

```
    package com.in28minutes.files;
    import java.nio.file.Files;
    import java.nio.file.Paths;
    import java.io.IOException;

    public class FileReadRunner {
        public static void main(String[] args) throws IOException {
            Path pathFileToRead = Paths.get("./resources/data.txt");
            List<String> lines = Files.readAllLines(pathFileToRead);
            System.out.println(lines);
        }
    }
```

Console Output

[123.122, asdfghjkl, Apple, Bat, Cat]

`Files.readAllLines(pathFileToRead)` makes it easy to read content of a file to list of String values.

Snippet-8 : Streamed File Read

`Files.readAllLines(pathFileToRead)` makes it easy to read content of a file. However, streaming is a better options when reading large files or when less memory is available.

./resources/data.txt

```
123.122

asdfghjkl

Apple

Bat

Cat
```

FileReadRunner.java

```java
package com.in28minutes.files;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;

public class FileReadRunner {
    public static void main(String[] args) throws IOException {
        Path pathFileToRead = Paths.get("./resources/data.txt");
        //List<String> lines = Files.readAllLines(pathFileToRead);
        //System.out.println(lines);
        Files.lines(pathFileToRead).forEach(System.out::println);
    }
}
```

Console Output

123.122

asdfghjkl

Apple

Bat

Cat

`Files.lines(pathFileToRead)` returns a stream which can be consumed as needed.

Snippet-9 : Printing file contents in lower-case

We can use `map` function to map to `String::toLowerCase`

./resources/data.txt

```
123.122

asdfghjkl

Apple

Bat

Cat
```

FileReadRunner.java

```java
package com.in28minutes.files;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;

public class FileReadRunner {
    public static void main(String[] args) throws IOException {
        Path pathFileToRead = Paths.get("./resources/data.txt");
        //Files.lines(pathFileToRead).forEach(System.out::println);
        Files.lines(pathFileToRead)
             .map(String::toLowerCase)
             .forEach(System.out::println);
    }
}
```

Console Output

123.122

asdfghjkl

apple

bat

cat

Snippet-9 : Filtering file contents

You can also filter file content using the `filter` method.

FileReadRunner.java

```
package com.in28minutes.files;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;

public class FileReadRunner {
    public static void main(String[] args) throws IOException {
        Path pathFileToRead = Paths.get("./resources/data.txt");
        //Files.lines(pathFileToRead).forEach(System.out::println);
        //Files.lines(pathFileToRead).map(String::toLowerCase).forEach(System.out::println);
        Files.lines(pathFileToRead)
             .map(String::toLowerCase)
             .filter(str -> str.contains("a"))
             .forEach(System.out::println);
    }
}
```

Console Output

asdfghjkl

apple

bat

cat

Snippet-10 : Writing to a file

`Files.write` can be used to write to a file.

FileWriteScanner.java

```
package com.in28minutes.files;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;

public class FileWriteRunner {
    public static void main(String[] args) throws IOException {
        Path pathFileToWrite = Paths.get("./resources/file-write.txt");
        List<String> list = List.of("Apple", "Boy", "Cat", "Dog", "Elephant");
        Files.write(pathFileToWrite, list);
    }
}
```

./resources/file-write.txt

```
Apple

Boy

Cat

Dog

Elephant
```

Concurrency : Advanced Topics

Let's create a simple counter.

Snippet-1: Atomic Operations : Counter

Counter.java

```
package com.in28minutes.concurrency;

public class Counter {
    private int i = 0;

    public void increment() {
        i++;
    }

    public int getI() {
        return i;
    }
}
```

ConcurrencyRunner.java

```
package com.in28minutes.concurrency;

public class ConcurrencyRunner {
    public static void main(String[] args) {
        Counter counter = new Counter();
        counter.increment();
        counter.increment();
        counter.increment();
        System.out.println(counter.get());
        }
    }
```

Snippet-1 Explained

Quite straightforward!

Counter `increment()` method is NOT Atomic!

Let's look at the code. It seemingly involves just one operation.

```
    public void increment() {
        i++;
    }
```

The operation `i++` actually involves the following steps:

- Step 1. Read the value of `i` from memory into the CPU registers
- Step 2. Increment the value in the CPU registers
- Step 3. Store the incremented value back into the memory location of `i`

`i++` is not an atomic operation.

What if `increment` is not atomic?

Let's take an example of two Threads, `T1` and `T2` running in `ConcurrencyRunner` access a single `Counter` object instance.

Let's say the threads concurrently invoke the `increment` and `getI` .

Assume the initial value of `i` is `15` . Let's take a closer look at a possible scenario:

- `T1` calls `increment()` first, and successfully completes Step 1 of `i++` . Gets a value of 15.
- The scheduler switches threads to `T2` .
- `T2` calls `increment()` first, and successfully completes Step 1 of `i++` .Gets a value of 15.
- The scheduler switches to `T1` . `T1` resumes execution of `increment()` , and completes steps 2 and 3 of `i++` . Completes execution of `increment()` . Value of `i` is over-written with `16` .
- The scheduler switches to `T2` . In it's CPU register context, i is `15` , not `16` . `T2` resumes execution of `increment()` , and completes steps 2 and 3 of `i++` . Completes execution of `increment()` .Value of `i` in the `Counter` instance is over-written with `16` .

Ideally, the final value of `i` after two `increment` s should have been `17` . This would result when the operations run serially one after the other.

This scenario, where the result of a concurrent computation (involving a sequence of operations) depends on the relative order of execution of those operations by the threads involved, is called a **race condition**.

There is a popular English saying: "There is many a slip, between the cup and the lip". This refers to the fact that anything can happen between the time when we hold a cup of tea in our hands, and the time when we actually get to take a sip of the tea.

This definitely rings true here.

The increment operation is not actually not as smooth as it seems. because it is not atomic, slip-ups can and will often occur. This brings us to the concept of **Thread-Safety**.

A method is said to be thread-safe, if it can be run in a concurrent environment (involving several concurrent invocations by independent threads) without *race-conditions*.

Revisited : The `synchronized` Keyword

Adding the keyword `synchronized` to the signature of a `class` method makes it thread safe.

Snippet-2

Counter.java

```
package com.in28minutes.concurrency;

public class Counter {
    private int i = 0;

    synchronized public void increment() {
        i++;
    }

    public int getI() {
        return i;
    }
}
```

Snippet-2 Explained

After adding `synchronized` keyword to the method `increment` , only one thread will be able to execute the method, at a time. Hence, race condition is avoided.

Snippet-3 : less concurrency

`synchronized` keyword make the code thread safe. However, it causes all other threads to wait. This can result in performance issues. Let's look at an example:

BiCounter.java

```
package com.in28minutes.concurrency;

public class BiCounter {
    private int i = 0;
    private int j = 0;

    synchronized public void incrementI() {
        i++;
    }

    synchronized public void incrementJ() {
        j++;
    }

    public int getI() {
        return i;
    }

    public int getJ() {
        return j;
    }
}
```

ConcurrencyRunner.java

```
package com.in28minutes.concurrency;
```

```
public class ConcurrencyRunner {
    public static void main(String[] args) {
        BiCounter counter = new BiCounter();
        counter.incrementI();
        counter.incrementJ();
        counter.incrementI();
        System.out.println(counter.get());
    }
}
```

Snippet-3 Explained

Both `incrementI()` and `incrementJ()` of `class BiCounter` are `synchronized`. Therefore, at any given time, at most one thread can execute either of these methods! Which means that, while a thread `T1` is executing `counter.incrementI()` within `ConcurrencyRunner.main()`, another thread `T2` **is not allowed to execute** `counter.incrementJ()`!

Just imagine, if there are a total of `12` threads wanting to increment `counter`. When one thread is running, the other `11` have to wait!

Synchronization With Locks

Let's look at another synchronization option - `Locks`

Snippet-4: BiCounter With Locks

BiCounterWithLocks.java

```
package com.in28minutes.concurrency;
import java.util.concurrent.locks.ReentrantLock;

public class BiCounterWithLocks {
    private int i = 0;
    private int j = 0;
    private Lock LockForI = new ReentrantLock();
    private Lock LockForJ = new ReentrantLock();

    public void incrementI() {
        lockForI.lock();
        i++;
        lockForI.unlock();
    }

    public void incrementJ() {
        lockForJ.lock();
        j++;
        lockForJ.unlock();
    }

    public int getI() {
        return i;
    }

    public int getJ() {
        return j;
    }
}
```

Snippet-4 Explained

`i++` and `j++` are the pieces of code to protect. We use two locks lockForI and lockForJ. If a thread wants to execute `i++`, it needs to first get a lock to it - implemented using `lockForI.lock()`. Once it performs the operation, it can release the lock `lockForI.unlock()` .

The `Lock` s `lockForI` and `lockForJ` are totally independent of each other. Therefore, a thread `T2` can execute `j++` within `incrementJ()` , at the same time that thread `T1` is executing `i++` within `incrementI()` .

Atomic Classes

The operation `i++` , small though it might seem, gave us quite a bit headache!

If a seemingly minute operation might need so much worrying about, imagine writing concurrent data structures that manipulate linked lists with multiple link operations!

It would be really nice if someone could take care of these tiny operations for us, otherwise we would have hard time finding out what code to definitely lock, and what code need not be!

Java addresses this issue for basic data types, by providing a few classes that are inherently thread-safe. A good example is `AtomicInteger` , which is a wrapper around the `int` primitive type.

Snippet-5 : AtomicInteger

BiCounterWithAtomicInteger.java

```
package com.in28minutes.concurrency;
import java.util.concurrent.atomic.AtomicInteger;

public class BiCounterWithAtomicInteger {
    private AtomicInteger i = new AtomicInteger();
    private int AtomicInteger = new AtomicInteger();

    public void incrementI() {
        i.incrementAndGet();
    }

    public void incrementJ() {
        j.incrementAndGet();
    }

    public int getI() {
        return i.get();
    }

    public int getJ() {
        return j.get();
    }
}
```

Snippet-5 Explained

`incrementAndGet` is atomic. So, `BiCounterWithAtomicInteger` does not need to worry about synchronization.

Concurrent Collections

Java gave us a ready-made solution for thread-safe primitive data, with wrappers such as `AtomicInteger` . The reasons this approach worked for `int` were:

- Simple, small-sized underlying type
- Wide-spread potential usage

How about collections?

Java provides classes like `Vector` (synchronized version of `ArrayList`) which can provide thread safety. But, these inherit the problems with using synchronized.

What are other options?

Snippet-6 : Need For ConcurrentMap

The code within the `for` loop does a `get` and then a `put` . It is not thread safe.

MapRunner.java

```
    package com.in28minutes.collections;
    import java.util.HashMap;

    public class MapRunner {
        public static void main(String[] args) {
            String str = "Hello World";
            Map<Character, Integer> occurrences = new HashMap<>();
            char[] characters = str.toCharArray();
            for(char character:characters) {
                Integer count = occurrences.get(character);
                if(count == null) {
                    occurrences.put(character, 1);
                } else {
                    occurrences.put(character, count + 1);
                }
            }
        }
    }
```

Concurrent Collections provide atomic versions of operations such as those encountered above:

- If entry does not exist, then create and initialize
- If entry exists, then update

The `ConcurrentMap` `interface`

The `interface` `ConcurrentMap` has methods to implement compound operations. Such operations include:

```
V putIfAbsent(K key, V value);

V computeIfPresent(K key,
            BiFunction<? super K, ? super V, ? extends V> remappingFunction)
```

Snippet-7 : ConcurrentHashMap Logic - Stage 1

`LongAdder` provides an atomic `increment` method, which we are making use of to make the code a little more thread safe. However, different threads could be executing `occurrences.get()` and `occurrences.put()` in parallel. A race condition can still occur.

ConcurrentMapRunner.java

```
    package com.in28minutes.concurrency;
    import java.util.Map;
    import java.util.HashTable;
    import java.util.concurrent.atomic.LongAdder;

    public class ConcurrentMapRunner {
        public static void main(String[] args) {
                String str = "ABCD ABCD ABCD";
                for(char character:str.toCharArray()) {
                  LongAdder longAdder = occurances.get(character);
                  if(longAdder == null) {
                        longAdder = new LongAdder();
                    }
                    longAdder.increment();
                    occurances.put(character, longAdder);
                }
        }
    }
```

Console Output

[=2, A=3, B=3, C=3, D=2]

Snippet-8 : ConcurrentHashMap Logic - Stage 2

We can use the method `computeIfAbsent()` from the `collection` `ConcurrentHashMap` to reduce the code to a single, atomic operation.

ConcurrentMapRunner.java

```
package com.in28minutes.concurrency;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.LongAdder;

public class ConcurrentMapRunner {
    public static void main(String[] args) {
        ConcurrentMap<Character, LongAdder> occurances = new ConcurrentHashMap<>();

        String str = "ABCD ABCD ABCD";

        for(char character:str.toCharArray()) {
          occurances.computeIfAbsent(character, ch -> new LongAdder()).increment();
        }

        System.out.println(occurances);
  }
}
```

Console Output

[=2, A=3, B=3, C=3, D=2]

`ConcurrentHashMap`

In `HashTable` , all methods are synchronized.

In `ConcurrentHashMap` , data structure is organized into disjoint regions. Access methods use different `Locks` for different regions, reducing performance impact during concurrent access.

Concurrent Collections : Copy-On-Write Optimization

All values in Copy-On-Write collections are stored in an internal immutable (not-changeable) array. A new array is created if there is any modification to the collection.

Read operations are not synchronized. Only write operations are synchronized.

Copy on Write approach is used in scenarios where reads greatly out number write's on a collection.

`CopyOnWriteArrayList` & `CopyOnWriteArraySet` are implementations of this approach.

Copy on Write collections are typically used in Subject – Observer scenarios, where the observers very rarely change. Most frequent operations would be iterating around the observers and notifying them.

Snippet-9 : CopyOnWriteArrayList

CopyOnWriteArrayListRunner.java

```java
package com.in28minutes.concurrency;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArrayListRunner {
    public static void main(String[] args) {
        List<String> list = new CopyOnWriteArrayList<>();
        // Total of 3 threads doing add()'s, maybe in a separate method
        list.add("Ant");
        list.add("Bat");
        list.add("Cat");

        //Total of 10000 threads looping on get(), again, in a separate method
        for(String element:list) {
            System.out.println(element);
        }
    }
}
```

Snippet-9 Explained

`CopyOnWriteArrayList.add()` method is a `synchronized` method. And the copy-on-write algorithm ensures that the copying is performed in a thread-safe manner, after which the write is done on a separate copy, while the `get()` 's continue on the original array. Once the `add()` is done, the collection starts using the new array, and discards the old one. This strategy continues for the lifetime of the program.

The `CopyOnWriteArrayList.get` method is NOT `synchronized` , since on the array that the reads work, there will be no direct write through an `add()` .

Copy-On-Write collections should only be used for the specific usage scenarios, viz., very large number of data structure traversals (data element reads only), compared to mutations (data element insertions/deletions/modifications). In this way, high concurrency is achieved for traversals.