JAVA SE TRAINING

Introduction

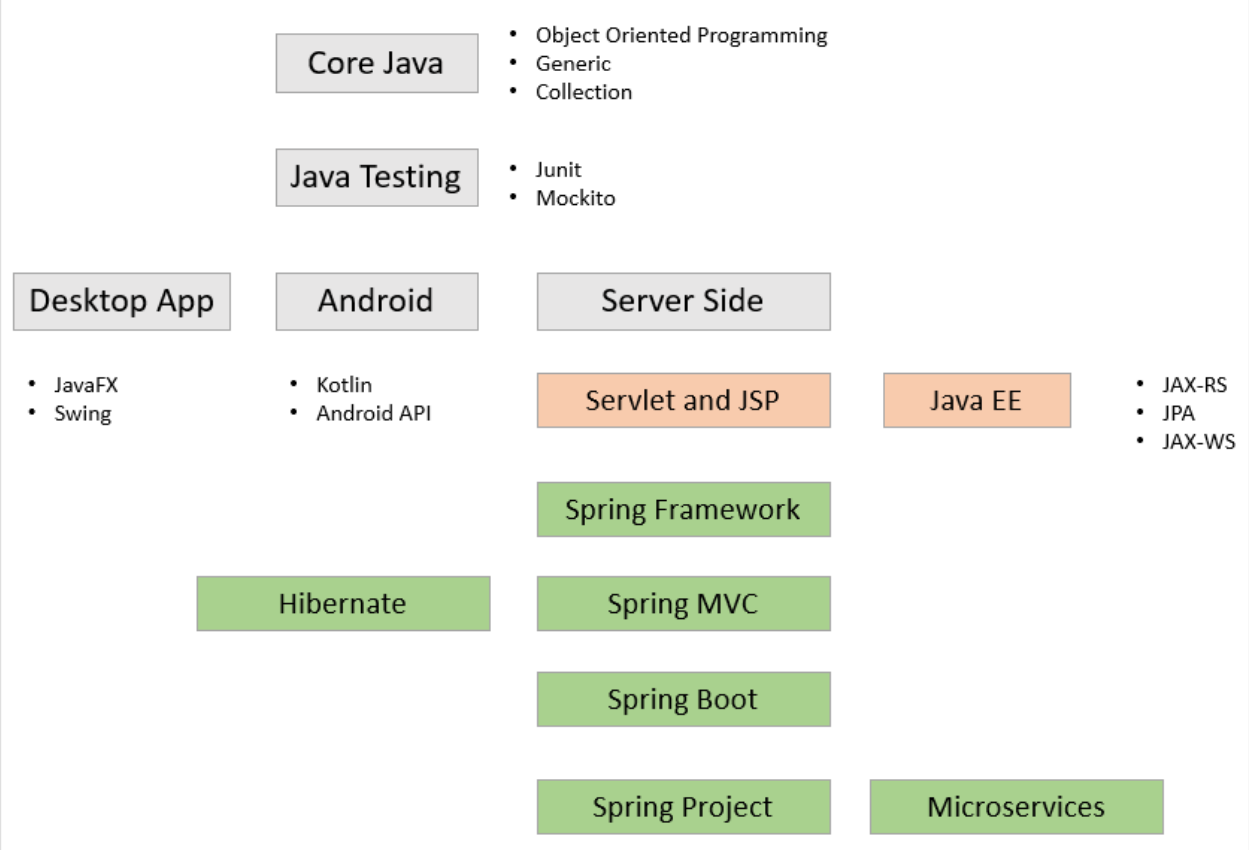
- 1.1 Introduction to Java SE Advanced Topic
- 1.2 Tools and Setup
- 1.3 Introduction to IntelliJ Idea

1.1 Introduction to Java SE Advanced Topic

In this module we will learn the Java SE Advanced learning path, history of Java SE and tools to use during the course.

Java Learning Path

The following diagram shows the choices of routes to learning and mastering java. Not all routes are compulsory to follow, choose only the one that meet your work expectation.



Others related tools to learn

- Design patterns
- Git

- Build Systems (Maven, Gradle, CICD)
- Microservices
- Data structures / algorithms

Release Table

The Java language has undergone several changes since JDK 1.0 as well as numerous additions of classes and packages to the standard library. Regarding Oracle Java SE Support Roadmap, version 18 is that latest versions, and versions 17, 11 and 8 are the currently supported long-term support (LTS) versions.

Version	Release Date	End of Free Public Updates
JDK Beta	1995	
JDK 1.0	Jan 1996	
JDK 1.1	Feb 1997	
J2SE 1.2	Dec 1998	
J2SE 1.3	May 2000	
J2SE 1.4	Feb 2002	
JSE 5	Sep 2004	
JSE 6	Apr 2013	
JSE 7	Jul 2011	
JSE 8 (LTS)	Mar 2014	March 2022
JSE 9	Sep 2017	
JSE 10	Mar 2018	
JSE 11 (LTS)	Sep 2018	Oct 2024
JSE 12	Mar 2019	
JSE 13	Sep 2019	
JSE 14	Mar 2020	
JSE 15	Sep 2020	
JSE 16	Mar 2021	
JSE 17 (LTS)	Sep 2021	Sep 2027
JSE 18	**Mar 2022**	
JSE 19	Sep 2022	
JSE 20	Mar 2023	
JSE 21 (LTS)	Sep 2023	Sep 2028

Oracle JDK vs. Open JDK

OpenJDK is a free and open-source implementation of the Java SE Platform Edition. It was initially released in 2007 as the result of the development that Sun Microsystems started in 2006. We should emphasize that OpenJDK is an official reference implementation of a Java Standard Edition since version SE 7.

	Oracle JDK	Open JDK
Release & Support	Provide LTS	No LTS
Licenses	License / Commercial	Free
Performance	More stable	Stable
Features	More features	Less features
Popularity	Less Popular	More Popular

1.2 Tools and Setup

In this course, we will use the following tools:

- JSE 17
- IntelliJ Idea
- MySQL 5.7
- Apache Tomcat 9.x
- SQL Yog
- Git Batch

Although the specific tools mention above, its not necessarily the ultimate choice. Here are several choices for every category.

Java SE

It's recommended to use at least JSE versions of 8.

IDE

Other options for IDE are as follows:

- Eclipse 2022.01
- Net Beans 13
- Visual Studio Code

Database Management

Other options for database management tool are as follows:

- PHP MyAdmin
- DBeaver
- MySQL Workbench
- HeidiSQL
- Navicat for MySQL
- Aqua Data Studio

Application Server

Other options for application server are as follows:

- Jetty
- Glass Fish
- WildFly
- Apache TomEE
- Apache Geronimo
- Red Hat JBoss
- IBM WebSphere
- Oracle Web Logic

Terminal

Other options for terminal tool are as follows:

- DOS
- Power Shell
- Cmder

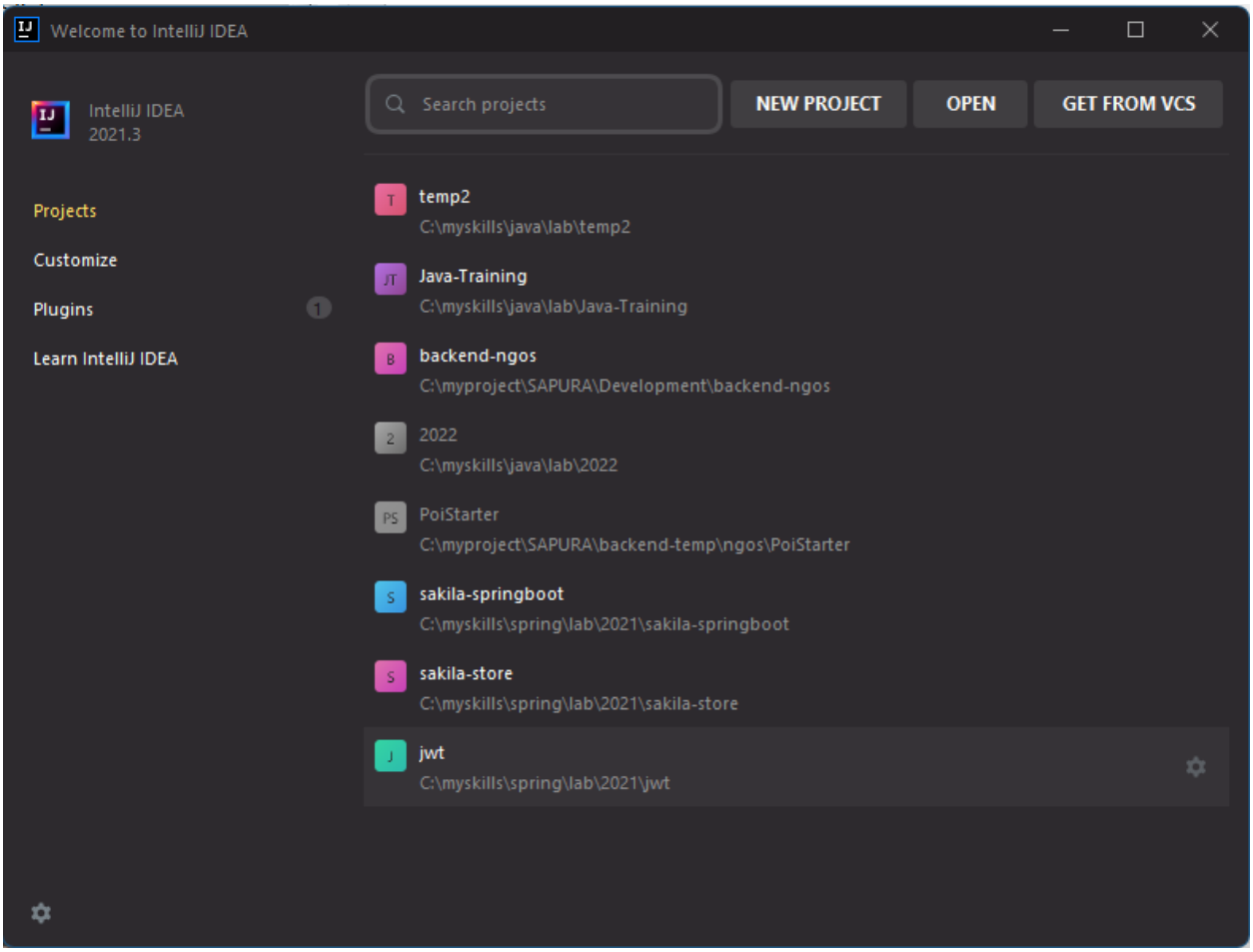
1.3 Introduction to JetBrains IntelliJ IDEA

The instructor will guide you how to use IntelliJ Idea in the following topics:

- Create a new project
- Plugins installation
- IDE Setting
- Navigation
- Short cut key
- Tip and trick

Welcome Page

- Create a new project here
- Open existing project
- Clone existing Git project
- Install / remove plugins



Learning the Basics

Overview

In this chapter, we will be executing programs that do not have the typical linear flow that we have seen so far. You will first learn to use if, else, else if, and switch-case statements to control the flow of your programs. You will practice running for, while, and do-while loops in order to perform repetitive tasks in Java, and how to pass command-line arguments to modify how programs run. By the end of this chapter, you will be able to implement immutable, static (global) variables, alongside Java's variable type inference mechanism.

Introduction

Business applications have lots of special-case conditions. Such conditions may include finding changes in allocation rules starting at a particular year, or handling different types of employees differently based on their designation. To code for such special cases, you will require conditional logic. You basically tell the computer to perform a set of actions when a particular condition is met.

Before we delve into advanced Java topics, you need to know the basics of Java syntax. While some of this material might seem simple, you'll find you need to use the techniques and syntax shown in this chapter repeatedly in your applications.

As you've seen in *Chapter 1*, *Getting Started*, Java's syntax borrows heavily from C and C++. That's true for conditional statements that control the flow of your programs as well. Java, like most computer languages, allows you to do this. This chapter covers the basic syntax of the Java language, especially ways in which you can control the flow of your applications.

This chapter, and the next one on object-oriented programming, will give you a good working knowledge of how Java programs work. You'll be able to take on more advanced APIs and topics. Work your way through this basic material, and you will be ready to move on to the more complex code to come.

Controlling the Flow of Your Programs

Imagine paying a bill from your e-wallet. You will only be able to make the payment if the credit balance in your e-wallet is greater than or equal to the bill amount. The following flowchart shows a simple logic that can be implemented:

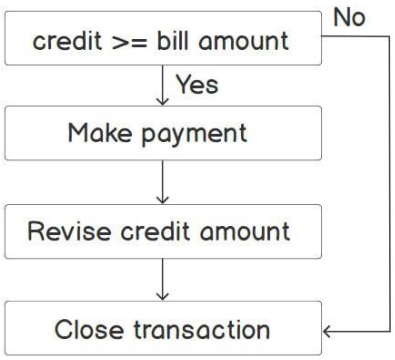


Figure 2.1: A representative flow chart for an if-else statement

Here, the credit amount dictates the course of action of the program. To facilitate such scenarios, Java uses the `if` statement.

With the `if` statement, your application will execute a block of code *if* (and only if) a particular condition is true. In the following code, if the `happy` variable is `true`, then the block of code immediately following the `if` statement will execute. If the `happy` variable is not `true`, then the block of code immediately following the `if` statement will not execute.

```
boolean happy = true;// initialize a Boolean variable as true
if (happy) //Checks if happy is true
    System.out.println("I am happy.");
```

Exercise 1: Creating a Basic if Statement

In most software industries, you are only working on a module of the code, and you might already know the value stored in a variable. You can use `if` statements and `print` statements in such cases. In this exercise, use an `if` statement to check if the values of variables assigned are `true` or `false`:

1. Create a directory for examples from this chapter and others. Name the folder `sources`.
2. In IntelliJ, select `File` -> `New` -> `Project` from the `File` menu.
3. In the `New Project` dialog box, select a `Java` project. Click `Next`.
4. Check the box to create the project from a template. Click on `Command Line App`. Click on `Next`.
5. Name the project `chapter02`.
6. For the project location, click the button with three dots (`...`) and then select the `sources` folder you created previously.
7. Delete the base package name so that this entry is left blank. You will use Java packages in the *Chapter 6*, *Libraries, Packages, and Modules*.
8. Click `Finish`.

   IntelliJ will create a project named `chapter02`, as well as a `src` folder inside `chapter02`. This is where your Java code will reside. IntelliJ also creates a class named `Main`:

```
public class Main {
    public static void main(String[] args) {
    // write your code here
    }
}
```

   Rename the class named `Main` to `Exercise01`. (We're going to create a lot of small examples in this chapter.)

9. Double-click in the text editor window on the word `Main` and then right-click it.
10. From the contextual menu, select `Refactor` | `Rename...`, enter `Exercise01`, and then press `Enter`.

    You will now see the following code:

```
public class Exercise01 {
    public static void main(String[] args) {
```

```
    // write your code here
    }
}
```

11. Within the `main()` method, define two Boolean variables, `happy` and `sad`:

```
12.      boolean happy = true;
    boolean sad = false;
```

13. Now, create two `if` statements, as follows:

```
14.  if (happy)
15.      System.out.println("I am happy.");
16.  // Usually put the conditional code into a block.
17.  if (sad) {
18.      // You will not see this.
19.      System.out.println("The variable sad is true.");
    }
```

The final code should look similar to this:

```
public class Exercise01 {
    public static void main(String[] args) {
        boolean happy = true;
        boolean sad = false;
        if (happy)
            System.out.println("I am happy.");
        // Usually put the conditional code into a block.
        if (sad) {
        // You will not see this.
```

```
            System.out.println("The variable sad is true.");
        }
    }
}
```

20. Click the green arrow that is just to the left of the text editor window that points at the class name `Exercise01`. Select the first menu choice, `Run Exercise01.main()`.
21. In the `Run` window, you'll see the path to your Java program, and then the following output:

```
I am happy.
```

The line `I am happy.` comes from the first `if` statement, since the `happy` Boolean variable is true.

Note that the second `if` statement does not execute, because the `sad` Boolean variable is false.

You almost always want to use curly braces to define the code block following an `if` condition. If you don't, you may find odd errors in your programs. For example, in the following code, the second statement, which sets the `i` variable to zero, will always get executed:

```
if (i == 5)
    System.out.println("i is 5");
    i = 0;
```

Unlike languages such as Python, indentation doesn't count in Java. The following shows what will actually execute with greater clarity:

```
if (i == 5) {
    System.out.println("i is 5");
}
i = 0;
```

The last line is always executed because it is outside the `if` statement after the curly braces closes.

Comparison Operators

In addition to Java's Booleans, you can use comparisons in conditional statements. These comparisons must form a Boolean expression that resolves to `true` or `false`. Comparison operators allow you to build Boolean expressions by comparing values. Java's main comparison operators include the following:

Operator	Name	Explanation
==	is equal to	a==b indicates that the value stored in variable a is equal to the value stored in b
!=	is not equal to	a!=b indicates that the value stored in variable a is not equal to the value stored in b
<	is less than	a<b indicates that the value stored in variable a is lesser than value stored in b
<=	is less than or equal to	a<=b indicates that the value stored in variable a is lesser than or equal to the value stored in b
>	is greater than	a>b indicates that the value stored in variable a is greater than the value stored in b
>=	is greater than or equal to	a>=b indicates that the value stored in variable a is greater than or equal to the value stored in b

Figure 2.2: The comparison operators in Java

The comparison operators such as `==` do not work the way you would expect for textual values. See the *Comparing Strings* section later in this chapter to see how to compare text values.

Note

A single equals sign, `=`, is used to assign a value. Two equals signs, `==`, is used to compare values. Therefore, generally, you never use `=` in a Boolean expression to check a condition.

Exercise 2: Using Java Comparison Operators

An online retail store provides free delivery only if the destination is within a 10-kilometer (km) radius of the store. Given the distance between the nearest store location and home, we can code this business logic with comparison operators:

1. In the `Project` pane in IntelliJ, right-click on the folder named `src`.
2. Choose `New` -> `Java Class` from the menu.
3. Enter `Exercise02` for the name of the new class.
4. Define the method named `main()`:

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

6. Inside the `main()` method, define the variables we'll use for comparisons:

```
7.    int maxDistance = 10;       // km
    int distanceToHome = 11;
```

8. Enter the following `if` statements after the variable declarations:

```
9.    if (distanceToHome > maxDistance) {
10.       System.out.println("Distance from the store to your home is");
11.       System.out.println("  more than " + maxDistance + "km away.");
12.       System.out.println("That is too far for free delivery.");
13.   }
14.   if (distanceToHome <= maxDistance) {
15.       System.out.println("Distance from the store to your home is");
16.       System.out.println("  within " + maxDistance + "km away.");
17.       System.out.println("You get free delivery!");
    }
```

The final code should look similar to the following: https://packt.live/32Ca9YS

18. Run the `Exercise02` program using the green arrow to the left.

In the `Run` window, you'll see the path to your Java program, and then the following output:

```
Distance from the store to your home is
  more than 10km away.
That is too far for free delivery.
```

Nested if Statements

Nesting implies embedding a construct within another code construct. You can nest `if` statements within any block of code, including the block of code that follows an `if` statement. Here is an example of how the logic in a nested `if` statement is evaluated:

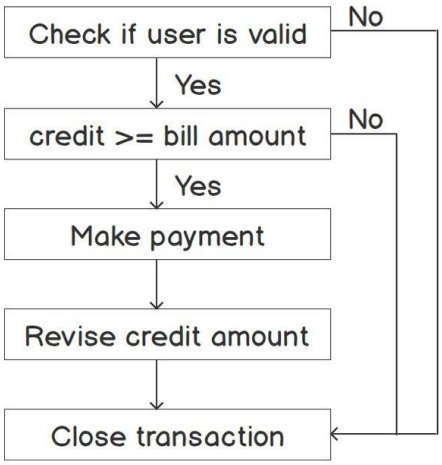


Figure 2.3: A representative flow chart for a nested if-else statement

Exercise 3: Implementing a Nested if Statement

In the following exercise, we will nest an `if` statement within another `if` statement to check if the speed of the vehicle is above the speed limit, and if so, whether it is above the finable speed:

1. Using the techniques from the previous exercise, create a new class named `Exercise03`.

2. Declare the `speed`, `speedForFine`, and `maxSpeed` variables with the values of `75`, `70`, and `60` respectively:

```
3.    public class Exercise03 {
4.        public static void main(String[] args) {
5.            int speed = 75;
6.            int maxSpeed = 60;
7.            int speedForFine = 70;
8.        }
    }
```

9. Create a nested `if` statement, where the outer `if` statement checks if the speed is greater than or equal to the maximum speed limit, and the inner loop checks if the speed is greater than or equal to the speed limit for a fine:

```
10.   // Nested if statements.
11.   if (speed >= maxSpeed) {
12.       System.out.println("You're over the speed limit!");
13.   if (speed >= speedForFine) {
14.       System.out.println("You are eligible for a fine!");
15.       }
    }
```

16. Run the `Exercise03` program using the green arrow to the left.

    In the `Run` window, you'll see the path to your Java program, and then the following output:

```
You're over the speed limit!
You are eligible for a fine!
```

    Note

Try changing the value of speed in the code and then running the program again. You will see how different speed values produce different outputs.

Branching Two Ways with if and else

An `else` statement following the code block for an `if` statement gets executed if the `if` statement condition is not true. You can also use `else if` statements to provide for an additional test.

The basic syntax is as follows:

```
if (speed > maxSpeed) {
    System.out.println("Your speed is greater than the max. speed limit");
} else if (speed < maxSpeed) {
    System.out.println("Your speed is less than the max. speed limit");
} else {
    System.out.println("Your speed is equal to the max. speed limit");
}
```

The third line (in the `else` block) will only print if neither of the first two lines (the `if` or `else if` code blocks) was true. Whatever the value of speed, only one of the lines will print.

Exercise 4: Using if and else Statements

A fair-trade coffee roaster offers a discount of 10% if you order more than 5 kg of whole coffee beans, and a discount of 15% if you order more than 50 kg. We'll code these business rules using `if`, `else if`, and `else` statements:

1. Using the techniques from the previous exercise, create a new class named `Exercise04`.
2. Enter the `main` method and declare the variables as follows:

```
3.    public static void main(String[] args) {
4.        int noDiscount = 0;
5.        int mediumDiscount = 10;      // Percent
6.        int largeDiscount = 15;
7.        int mediumThreshold = 5;      // Kg
8.        int largeThreshold = 50;
9.        int purchaseAmount = 40;
    }
```

10. Enter the following `if`, `else if`, and `else` statements:

```
11.   if (purchaseAmount >= largeThreshold) {
12.       System.out.println("You get a discount of " + largeDiscount +
    "%");
13.   } else if (purchaseAmount >= mediumThreshold) {
14.       System.out.println("You get a discount of " + mediumDiscount +
    "%");
15.   } else {
16.       // Sorry
17.       System.out.println("You get a discount of " + noDiscount +
    "%");
    }
```

Notice that we check against the largest threshold first. The reason for this is that a value greater than or equal to `largeThreshold` will also be greater than or equal to `mediumThreshold`.

Note

The full source code for this exercise can be found at: https://packt.live/33UTu35.

18. Run the `Exercise04` program using the green arrow to the left.

    In the **Run** window, you'll see the path to your Java program, and then the following output:

```
You get a discount of 10%
```

Using Complex Conditionals

Java allows you to create complex conditional statements with logical operators. Logical operators are generally used on only Boolean values. Here are some of the logical operators available in Java:

- `AND (&&)`: `a` && `b` will be evaluated to `true` if both a and b are `true`
- `OR (||)`: `a` || `b` will be evaluated to `true` if either a or b, or both are `true`
- `NOT (!)`: `!a` be evaluated to true if `a` is `false`

Use the conditional operators to check more than one condition in an `if` statement. For example, the following shows an `if` statement where both conditions must be true for the overall `if` statement to execute:

```
boolean red = true;
boolean blue = false;
if ((red) && (blue)) {
    System.out.println("Both red AND blue are true.");
}
```

In this case, the overall expression resolves to `false`, since the `blue` variable is false, and the `print` statement will not execute.

Note

Always use parentheses to make your conditionals clear by grouping the conditions together.

You can also check if either, or both, of the expressions are true with the `||` operator:

```
boolean red = true;
boolean blue = false;
if ((red) || (blue)) {
    System.out.println("Either red OR blue OR both are true.");
}
```

In this case, the overall expression resolves to true, since at least one part is true. Therefore, the `print` statement will execute:

```
boolean blue = false;
if (!blue) {
    System.out.println("The variable blue is false");
}
```

The value of `blue` is initialized to `false`. Since we are checking the NOT of the blue variable in the `if` statement, the `print` statement will execute. The following exercise shows how we can use logical operators.

Exercise 5: Using Logical Operators to Create Complex Conditionals

This exercise shows an example of each of the conditional operators previously described. You are writing an application that works with data from a fitness tracker. To accomplish this, you need to code a check against normal heart rates during exercise.

If a person is 30 years old, a normal heart rate should be between 95 beats per minute (bpm) and 162 bpm. If the person is 60 years old, a normal heart rate should be between 80 and 136 bpm.

Use the following steps for completion:

1. Using the techniques from the previous exercise, create a new class named `Exercise05` in the `main` method and declare variables.

```
2.    public static void main(String[] args) {
3.        int age = 30;
4.        int bpm = 150;
    }
```

5. Create an `if` statement to check the heart rate of a 30-year old person:

```
6.    if (age == 30) {
7.        if ((bpm >= 95) && (bpm <= 162)) {
8.            System.out.println("Heart rate is normal.");
9.        } else if (bpm < 95) {
10.           System.out.println("Heart rate is very low.");
11.       } else {
12.           System.out.println("Heart rate is very high.");
      }
```

We have nested conditionals to check the allowable range for 30-year-olds.

13. Create an `else if` statement to check the heart rate of a 60-year old person:

```
14.   } else if (age == 60) {
15.       if ((bpm >= 80) && (bpm <= 136)) {
16.           System.out.println("Heart rate is normal.");
17.       } else if (bpm < 80) {
18.           System.out.println("Heart rate is very low.");
19.       } else {
20.           System.out.println("Heart rate is very high.");
21.       }
```

```
}
```

We have nested conditionals to check the allowable range for 60-year-olds.

22. Run the `Exercise05` program using the green arrow to the left.

    In the `Run` window, you'll see the path to your Java program, and then the following output:

```
Heart rate is normal.
```

23. Change `age` to `60` and re-run the program; your output should be as follows:

```
Heart rate is very high.
```

Note

The full source code for this exercise can be found at: https://packt.live/2W3YAHs.

Using Arithmetic Operators in an if Condition

You can use arithmetic operators as well in Boolean expressions, as shown in `Example01.java`:

```
public class Example01 {
    public static void main(String[] args) {
        int x = 2;
        int y = 1;
        if ((x + y) < 5) {
            System.out.println("X added to Y is less than 5.");
        }
    }
}
```

The output in this case would be as follows:

```
X added to Y is less than 5
```

Here, the value of `(x + y)` is calculated, and then the result is compared to `5`. So, since the result of `x` added to `y` is `3`, which is less than `5`, the condition holds true. Therefore, the `print` statement is executed. Now that we have seen the variations of the `if else` statement, we will now see how we can use the ternary operator to express the `if else` statements.

The Ternary Operator

Java allows a short-hand version of an `if else` statement, using the ternary (or three-part) operator, `?:`. This is often used when checking variables against an allowed maximum (or minimum) value.

The basic format is: *Boolean expression* `?` *true block* `:` *false block*, as follows:

```
x = (x > max) ? max : x;
```

The JVM resolves the `(x > max)` Boolean expression. If true, then the expression returns the value immediately after the question mark. In this case, that value will be set into the `x` variable since the line of code starts with an assignment, `x =`. If the expression resolves to false, then the value after the colon, `:`, is returned.

Exercise 6: Using the Ternary Operator

Consider the minimum height requirement for a roller coaster to be 121 centimeters (cm). In this exercise, we will check for this condition using the ternary operator. Perform the following steps:

1. Using the techniques from the previous exercise, create a new class named `Exercise06`.
2. Declare and assign values to the `height` and `minHeight` variables. Also, declare a string variable to print the output message:

```
3.    public static void main(String[] args) {
4.        int height = 200;
5.        int minHeight = 121;
      String result;
```

6. Use the ternary operator to check the minimum height requirement and set the value of `result`:

```
7.    result = (height > minHeight) ? "You are allowed on the ride" :
    "Sorry you do not meet the height requirements";
8.    System.out.println(result);
    }
```

So, if height is greater than `minHeight`, the first statement will be returned (`You are allowed on the ride`). Otherwise, the second statement will be returned (`Sorry you do not meet the height requirements`).

Your code should look similar to this:

```
public class Exercise06 {
    public static void main(String[] args) {
        int height = 200;
        int minHeight = 121;
        String result;
        result = (height > minHeight) ? "You are allowed on the
ride" :          "Sorry you do not meet the height requirements";
        System.out.println(result);
    }
}
```

9. Run the `Exercise06` program.

In the `Run` window, you'll see the path to your Java program, and then the following output:

```
You are allowed on the ride
```

Equality Can Be Tricky

Java decimal types such as `float` and `double` (and the object versions, `Float` and `Double`) are not stored in memory in a way that works with regular equality checks.

When comparing decimal values, you normally need to define a value that represents what you think is close enough. For example, if two values are within *.001* of each other, then you may feel that is close enough to consider the values as equal.

Exercise 7: Comparing Decimal Values

In this exercise, you'll run a program that checks if two double values are close enough to be considered equal:

1. Using the techniques from the previous exercise, create a new class named `Exercise07`.
2. Enter the following code:

```
3.   public class Exercise07 {
4.       public static void main(String[] args) {
5.           double a = .6 + .6 + .6 + .6 + .6 + .6;
6.           double b = .6 * 6;
7.           System.out.println("A is " + a);
8.           System.out.println("B is " + b);
9.           if (a != b) {
10.              System.out.println("A is not equal to B.");
11.          }
```

```
12.             // Check if close enough.
13.             if (Math.abs(a - b) < .001) {
14.                 System.out.println("A is close enough to B.");
15.             }
16.         }
    }
```

The `Math.abs()` method returns the absolute value of the input, making sure the input is positive.

We will learn more about the `Math` package in *Chapter 6*, *Libraries, Packages, and Modules*.

17. Run the `Exercise07` program using the green arrow to the left.

    In the run window, you'll see the path to your Java program, and then the following output:

```
A is 3.6
B is 3.5999999999999996
A is not equal to B.
A is close enough to B.
```

    Note how `a` and `b` differ due to the internal storage for the double type.

    Note

    For more on how Java represents floating-point numbers, see https://packt.live/2VZdaQy.

Comparing Strings

You cannot use `==` to compare two strings in Java. Instead, you need to use the `String` class' `equals` method. This is because `==` with `String` objects just checks whether they are the exact same object. What you'll normally want is to check if the string values are equal:

```
String cat = new String("cat");
String dog = new String("dog");
if (cat.equals(dog)) {
    System.out.println("Cats and dogs are the same.");
}
```

The `equals` method on a `String` object called `cat` returns true if the passed-in `String`, `dog`, has the same value as the first `String`. In this case, these two strings differ. So, the Boolean expression will resolve to false.

You can also use literal strings in Java, delineating these strings with double quotes. Here's an example:

```
if (dog.equals("dog")) {
    System.out.println("Dogs are dogs.");
}
```

This case compares a `String` variable named `dog` with the literal string `"dog"`.

`Example09` shows how to call the equals method:

Example09.java

```
15              if (dog.equals(dog)) {
16              System.out.println("Dogs are dogs.");
17          }
18
```

```
19          // Using literal strings
20          if (dog.equals("dog")) {
21              System.out.println("Dogs are dogs.");
22          }
23
24          // Can compare using a literal string, too.
25          if ("dog".equals(dog)) {
26              System.out.println("Dogs are dogs.");
```

https://packt.live/2BtrKGz

You should get the following output:

```
Cats and dogs are not the same.
Dogs are dogs.
Dogs are dogs.
Dogs are dogs.
```

Using switch Statements

The `switch` statement is similar to a group of nested `if-else-if` statements. With `switch`, you can choose from a group of values.

The basic syntax follows:

```
switch(season) {
    case 1: message = "Spring";
        break;
    case 2: message = "Summer";
```

```
        break;
    case 3: message = "Fall";
        break;
    case 4: message = "Winter";
        break;
    default: message = "That's not a season";
        break;
}
```

With the `switch` keyword, you place the variable to be checked. In this case, we're checking a variable called `season`. Each case statement represents one possible value for the `switch` variable (season). If the value of season is `3`, then the `case` statement that matches will be executed, setting the `message` variable to the String `Fall`.
The `break` statement ends the execution for that case.

The `default` statement is used as a catch-all for any unexpected value that doesn't match the defined cases. The best practice is to always include a `default` statement.
Let's see how to implement this logic in a program.

Exercise 8: Using switch

In this exercise, you'll run a program that maps a number to a season:

1. Using the techniques from the previous exercise, create a new class named `Exercise08`.
2. Enter in the `main()` method and set up these variables:

```
3.    public static void main(String[] args) {
4.        int season = 3;
5.        String message;
    }
```

6. Enter the following `switch` statement.

```
7.        switch(season) {
8.            case 1: message = "Spring";
9.                break;
10.           case 2: message = "Summer";
11.               break;
12.           case 3: message = "Fall";
13.               break;
14.           case 4: message = "Winter";
15.               break;
16.           default: message = "That's not a season";
17.               break;
    }
```

18. And enter a `println` statement to show us the results:

```
System.out.println(message);
```

Note

You can find the code for this exercise here: https://packt.live/35WXm58.

19. Run the `Exercise08` program using the green arrow to the left.

In the `Run` window, you'll see the path to your Java program, and then the following output:

```
Fall
```

Because the `season` variable is set to `3`, Java executes the `case` with `3` as the value, so in this case, setting the `message` variable to `Fall`.

Note

> There is no one rule for when to use a `switch` statement as opposed to a series of `if-else` statements. In many cases, your choice will be based on the clarity of the code. In addition, `switch` statements are limited in only having cases that hold a single value, while `if` statements can test much more complicated conditions.

Normally, you'll put a `break` statement after the code for a particular case. You don't have to. The code will keep executing from the start of the `case` until the next `break` statement. This allows you to treat multiple conditions similarly.

Exercise 9: Allowing Cases to Fall Through

In this exercise, you will determine a temperature adjustment for the porridge in *Goldilocks and the Three Bears*. If the porridge is too hot, for example, you'll need to reduce the temperature. If it's too cold, raise the temperature:

1. Using the techniques from the previous exercise, create a new class named `Exercise09`.
2. Enter in the `main()` method and set up these variables:

```
3.    public static void main(String[] args) {
4.        int tempAdjustment = 0;
5.        String taste = "way too hot";
    }
```

6. Next, enter the following `switch` statement:

```
7.        switch(taste) {
8.            case "too cold":    tempAdjustment += 1;
9.                break;
10.           case "way too hot": tempAdjustment -= 1;
11.           case "too hot":     tempAdjustment -= 1;
12.               break;
13.           case "just right":  // No adjustment
14.           default:
```

```
15.                 break;
    }
```

16. Print out the results:

```
    System.out.println("Adjust temperature: " + tempAdjustment);
```

17. Run the `Exercise09` program using the green arrow to the left.

    In the run window, you'll see the path to your Java program, and then the following output:

```
Adjust temperature: -2
```

    Look carefully at the `switch` statement. If the value of the taste variable is too cold, then increment the temperature by 1. If the value is too hot, decrement the temperature by 1. But notice there is no break statement, so the code keeps executing and adjusts the temperature down by another 1. This implies that if the porridge is too hot, the temperature is decremented by 1. If it's way too hot, it's decremented by 2. If the porridge is just right, there is no adjustment.

    Note

    Starting with Java 7, you can use Strings in switch statements. Prior to Java 7, you could not.

Using Java 12 Enhanced switch Statements

Java 12 offers a new form of the `switch` statement. Aimed at `switch` statements that are essentially used to determine the value of a variable, the new `switch` syntax allows you to assign a variable containing the result of the `switch`.

The new syntax looks like this:

```
int tempAdjustment = switch(taste) {
    case "too cold" ->    1;
    case "way too hot" -> -2;
```

```
    case "too hot" ->    -1;
    case "just right" -> 0;
    default -> 0;
};
```

This `switch` syntax does not use break statements. Instead, for a given case, only the code block after `->` gets executed. The value from that code block is then returned as the value from the switch statement.

We can rewrite the `Exercise09` example using the new syntax, as shown in the following exercise.

Note

IntelliJ needs a configuration to support Java 12 `switch` statements.

Exercise 10: Using Java 12 switch Statements

In this exercise, we will work on the same example as in the previous exercise. This time, though, we will implement the new switch case syntax that is made available by Java 12. Before we start with the program there, you'll have to make changes to the IntelliJ configuration. We will set that up in the initial few steps of the exercise:

1. From the `Run` menu, select `Edit` Configurations.
2. Click on `Edit` Templates.
3. Click on `Application`.
4. Add the following to the `VM` options:
   ```
   --enable-preview
   ```
5. Click `OK`.

   This turns on the IntelliJ support for Java 12 enhanced switch statements.

6. Using the techniques from the previous exercise, create a new class named `Exercise10`.
7. Enter in the `main()` method and set up this variable:

```
8.    public static void main(String[] args) {
9.        String taste = "way too hot";
    }
```

10. Define a `switch` statement as follows:

```
11.   int tempAdjustment = switch(taste) {
12.       case "too cold" ->    1;
13.       case "way too hot" -> -2;
14.       case "too hot" ->    -1;
15.       case "just right" -> 0;
16.       default -> 0;
    };
```

Note the semi-colon after `switch`. Remember, we are assigning a variable to a value with the whole statement.

17. Then print out the value chosen:

```
System.out.println("Adjust temperature: " + tempAdjustment);
```

18. When you run this example, you should see the same output as in the previous example:

```
Adjust temperature: -2
```

The full code is as follows:

```
public class Exercise10 {
    public static void main(String[] args) {
        String taste = "way too hot";
        int tempAdjustment = switch(taste) {
            case "too cold" ->    1;
```

```
            case "way too hot" -> -2;
            case "too hot" ->    -1;
            case "just right" -> 0;
            default -> 0;
        };
        System.out.println("Adjust temperature: " + tempAdjustment);
    }
}
```

Looping and Performing Repetitive Tasks

In this chapter, we cover using loops to perform repetitive tasks. The main types of loop are as follows:

- `for` loops
- `while` loops
- `do-while` loops

`for` loops repeat a block a set number of times. Use a `for` loop when you are sure how many iterations you want. A newer form of the `for` loop iterates over each item in a collection.

`while` loops execute a block while a given condition is true. When the condition becomes false, the `while` loop stops. Similarly, `do-while` loops execute a block and then check a condition. If true, the `do-while` loop runs the next iteration.

Use `while` loops if you are unsure how many iterations are required. For example, when searching through data to find a particular element, you normally want to stop when you find it.

Use a `do-while` loop if you always want to execute the block and only then check if another iteration is needed.

Looping with the for Loop

A `for` loop executes the same block of code for a given number of times. The syntax comes from the C language:

```
for(set up; boolean expression; how to increment) {
    // Execute these statements…
}
```

In the preceding code, we can see that:

- Each part is separated by a semicolon, (`;`).
- The `set up` part gets executed at the beginning of the entire for loop. It runs once.
- The `boolean expression` is examined at each iteration, including the first. So long as this resolves to true, the loop will execute another iteration.
- The `how to increment` part defines how you want a loop variable to increment. Typically, you'll add one for each increment.

The following exercise will implement a classic for loop in Java.

Exercise 11: Using a Classic for Loop

This exercise will run a `for` loop for four iterations, using the classic for loop syntax:

1. Using the techniques from the previous exercise, create a new class named `Exercise11`.
2. Enter a `main()` method and the following code:

```
3.    public static void main(String[] args) {
4.        for (int i = 1; i < 5; i++) {
5.            System.out.println("Iteration: " + i);
6.        }
    }
```

7. Run the `Exercise11` program using the green arrow to the left.

   In the `Run` window, you'll see the path to your Java program, and then the following output:

```
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
```

Here is how the program executes:

- `int i = 1` is the `for` loop set up part.
- The Boolean expression checked each iteration is `i < 5`.
- The how to increment part tells the `for` loop to add one to each iteration using the `++` operator.
- For each iteration, the code inside the parentheses executes. It continues like this until the Boolean expression stops being `true`.

In addition to the old classic `for` loop, Java also offers an enhanced for loop designed to iterate over collections and arrays.

We will cover arrays and collections in greater detail later in the book; for now, think of arrays as groups of values of the same data type stored in a single variable, whereas collections are groups of values of different data types stored in a single variable.

Exercise 12: Using an Enhanced for Loop

Iterating over the elements of arrays means that the increment value is always 1 and the start value is always 0. This allows Java to reduce the syntax of a form to iterate over arrays. In this exercise you will loop over all items in a `letters` array:

1. Using the techniques from the previous exercise, create a new class named `Exercise12`.
2. Enter a `main()` method:

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

4. Enter the following array:

```
String[] letters = { "A", "B", "C" };
```

Chapter 4, *Collections, Lists, and Java's Built-In APIs*, will cover the array syntax in greater depth. For now, we have an array of three `String` values, `A`, `B`, and `C`.

5. Enter an enhanced `for` loop:

```
6.    for (String letter : letters) {
7.        System.out.println(letter);
    }
```

Notice the reduced syntax of the `for` loop. Here, the variable letter iterates over every element in the letters array.

8. Run the `Exercise12` program using the green arrow to the left.

In the `Run` window, you'll see the path to your Java program, and then the following output:

```
A
B
C
```

Jumping Out of Loops with Break and Continue

A `break` statement, as we saw in the `switch` examples, jumps entirely out of a loop. No more iterations will occur.

A `continue` statement jumps out of the current iteration of the loop. Java will then evaluate the loop expression for the next iteration.

Exercise 13: Using break and continue

This exercise shows how to jump out of a loop using `break`, or jump to the next iteration using continue:

1. Using the techniques from the previous exercise, create a new class named `Exercise13`.
2. Enter a `main()` method:

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

    }
```

4. Define a slightly longer array of `String` values:

```
String[] letters = { "A", "B", "C", "D" };
```

5. Enter the following for loop:

```
6.    for (String letter : letters) {

    }
```

   This loop will normally iterate four times, once for each letter in the `letters` array. We'll change that though, with the next code.

7. Add a conditional to the loop:

```
8.    if (letter.equals("A")) {

9.        continue;    // Jump to next iteration

    }
```

   Using `continue` here means that if the current letter equals `A`, then jump to the next iteration. None of the remaining loop code will get executed.

10. Next, we'll print out the current letter:

```
System.out.println(letter);
```

    For all iterations that get here, you'll see the current letter printed.

11. Finish the `for` loop with a conditional using `break`:

```
12.   if (letter.equals("C")) {
13.       break;     // Leave the for loop
      }
```

    If the value of `letter` is `C`, then the code will jump entirely out of the loop. And since our array of letters has another value, `D`, we'll never see that value at all. The loop is done when the value of letter is `C`.

14. Run the `Exercise13` program using the green arrow to the left.

    In the `Run` window, you'll see the path to your Java program, and then the following output:

```
B
C
```

`Exercise13.java` holds the full example:

Note

Source code for Exercise 13 can be found at the following link: https://packt.live/2MDczAV.

Using the while Loop

In many cases, you won't know in advance how many iterations you need. In that case, use a `while` loop instead of a `for` loop.

A `while` loop repeats so long as (or *while*) a Boolean expression resolves to true:

```
while (boolean expression) {
    // Execute these statements…
}
```

Similar to a `for` loop, you'll often use a variable to count iterations. You don't have to do that, though. You can use any Boolean expression to control a `while` loop.

Exercise 14: Using a while Loop

This exercise implements a similar loop to *Exercise10*, which shows a `for` loop:

1. Using the techniques from the previous exercise, create a new class named `Exercise14`.
2. Enter a `main()` method:

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

4. Enter the following variable setting and `while` loop:

```
5.    int i = 1;
6.    while (i < 10) {
7.        System.out.println("Odd: " + i);
8.        i += 2;
    }
```

   Note how this loop increments the `i` variable by two each time. This results in printing odd numbers.

9. Run the `Exercise14` program using the green arrow to the left.

   In the `Run` window, you'll see the path to your Java program, and then the following output:

```
Odd: 1
Odd: 3
Odd: 5
Odd: 7
```

```
Odd: 9
```

Note

A common mistake is to forget to increment the variable used in your Boolean expression.

Using the do-while Loop

The `do-while` loop provides a variant on the `while` loop. Instead of checking the condition first, the `do-while` loop checks the condition after each iteration. This means with a `do-while` loop, you will always have at least one iteration. Normally, you will only use a `do-while` loop if you are sure you want the iteration block to execute the first time, even if the condition is false.

One example use case for the `do-while` loop is if you are asking the user a set of questions and then reading the user's response. You always want to ask the first question.

The basic format is as follows:

```
do {
    // Execute these statements…
} while (boolean expression);
```

Note the semicolon after the Boolean expression.

A `do-while` loop runs the iteration block once, and then checks the Boolean expression to see if the loop should run another iteration.

`Example17.java` shows a `do-while` loop:

```
public class Example17 {
    public static void main(String[] args) {
```

```
        int i = 2;
        do {
            System.out.println("Even: " + i);
            i += 2;
        } while (i < 10);
    }
}
```

This example prints out even numbers.

Note

You can use `break` and `continue` with `while` and `do-while` loops too.

Handling Command-Line Arguments

Command-line arguments are parameters passed to the `main()` method of your Java program. In each example so far, you've seen that the `main()` method takes in an array of `String` values. These are the command-line arguments to the program.

Command-line arguments prove their usefulness by giving you one way of providing inputs to your program. These inputs are part of the command line launching the program, when run from a Terminal shell window.

Exercise 15: Testing Command-Line Arguments

This exercise shows how to pass command-line arguments to a Java program, and also shows how to access those arguments from within your programs:

1. Using the techniques from the previous exercises, create a new class named `Exercise15`.
2. Enter the following code:

```
3.    public class Exercise15 {
4.        public static void main(String[] args) {
5.            for (int i = 0; i < args.length; i++) {
6.                System.out.println(i + " " + args[i]);
7.            }
8.        }
    }
```

This code uses a `for` loop to iterate over all the command-line arguments, which the `java` command places into the `String` array named `args`.

Each iteration prints out the position (`i`) of the argument and the value (`args[i]`). Note that Java arrays start counting positions from 0 and `args.length` holds the number of values in the `args` array.

To run this program, we're going to take a different approach than before.

9. In the bottom of the IntelliJ application, click on `Terminal`. This will show a command-line shell window.

   When using IntelliJ for these examples, the code is stored in a folder named `src`.

10. Enter the following command in the `Terminal` window:

    ```
    cd src
    ```

    This changes to the folder with the example source code.

11. Enter the `javac` command to compile the Java program:

    ```
    javac Exercise15.java
    ```

    This command creates a file named `Exercise15.class` in the current directory. IntelliJ normally puts these *.class* files into a different folder.

12. Now, run the program with the `java` command with the parameters you want to pass:

```
java Exercise15 cat dog wombat
```

In this command, `Exercise15` is the name of the Java class with the `main()` method, `Exercise15`. The values following `Exercise15` on the command line are passed to the `Exercise15` application as command-line arguments. Each argument is separated by a space character, so we have three arguments: *cat*, *dog*, and *wombat*.

13. You will see the following output:

```
14.  0 cat
15.  1 dog
2 wombat
```

The first argument, at position `0` in the `args` array, is `cat`. The argument at position `1` is `dog`, and the argument at position `2` is `wombat`.

Note

The `java` command, which runs compiled Java programs, supports a set of command-line arguments such as defining the available memory heap space. See the Oracle Java documentation at https://packt.live/2BwqwdJ for details on the command-line arguments that control the execution of your Java programs.

Converting Command-Line Arguments

Command-line arguments appear in your Java programs as String values. In many cases, though, you'll want to convert these *String values* into numbers.

If you are expecting an integer value, you can use `Integer.parseInt()` to convert a `String` to an `int`.

If you are expecting a double value, you can use `Double.parseDouble()` to convert a `String` to a `double`.

Exercise 16: Converting String to Integers and Doubles

This exercise extracts command-line arguments and turns them into numbers:

1. Using the techniques from the previous exercises, create a new class named `Exercise16`.
2. Enter the `main()` method:

```
3.    public class Exercise16 {
4.        public static void main(String[] args) {
5.        }
    }
```

6. Enter the following code to convert the first argument into an `int` value:

```
7.        if (args.length > 0) {
8.            int intValue = Integer.parseInt(args[0]);
9.            System.out.println(intValue);
      }
```

This code first checks if there is a command-line argument, and then if so, converts the `String` value to an `int`.

10. Enter the following code to convert the second argument into a `double` value:

```
11.       if (args.length > 1) {
12.           double doubleValue = Double.parseDouble(args[1]);
13.           System.out.println(doubleValue);
      }
```

This code checks if there is a second command-line argument (start counting with 0) and if so, converts the `String` to a `double` value.

14. Enter the `javac` command introduced in *Chapter 1*, *Getting Started*, to compile the Java program:

```
javac Exercise16.java
```

This command creates a file named `Exercise16.class` in the current directory.

15. Now, run the program with the `java` command:

```
java Exercise16 42 65.8
```

You will see the following output:

```
42
65.8
```

The values printed out have been converted from String values into numbers inside the program. This example does not try to catch errors, so you have to enter the inputs properly.

Note

Both `Integer.parseInt()` and `Double.parseDouble()` will throw `NumberFormatException` if the passed-in String does not hold a number. See *Chapter 5*, *Exceptions*, for more on exceptions.

Diving Deeper into Variables — Immutability

Immutable objects cannot have their values modified. In Java terms, once an immutable object is constructed, you cannot modify the object.

Immutability can provide a lot of advantages for the JVM, since it knows an immutable object cannot be modified. This can really help with garbage collection. When writing programs that use multiple threads, knowing an object cannot be modified by another thread can make your code safer.

In Java, `String` objects are immutable. While it may seem like you can assign a `String` to a different value, Java actually creates a new object when you try to change a `String`.

Comparing Final and Immutable

In addition to immutable objects, Java provides a `final` keyword. With `final`, you cannot change the object reference itself. You can change the data within a `final` object, but you cannot change which object is referenced.

Contrast `final` with immutable. An immutable object does not allow the data inside the object to change. A `final` object does not allow the object to point to another object.

Class(template/blueprint) vs Object(Instance of the class)

Using Static Values

A *static* variable is common to all instances of a class. This differs from instance variables that apply to only one instance, or object, of a class. For example, each instance of the `Integer` class can hold a different `int` value. But, in the `Integer` class, `MAX_VALUE` and `MIN_VALUE` are static variables. These variables are defined once for all instances of integers, making them essentially global variables.

Note

Chapter 3, *Object-Oriented Programming*, delves into classes and objects.

Static variables are often used as constants. To keep them constant, you normally want to define them as `final` as well:

```
public static final String MULTIPLY = "multiply";
```

Note

By convention, the names of Java constants are all uppercase.

`Example20.java` defines a constant, `MULTIPLY`:

```
public class Example20 {
    public static final String MULTIPLY = "multiply";
    public static void main(String[] args) {
        System.out.println("The operation is " + MULTIPLY);
    }
}
```

Because the `MULTIPLY` constant is a final value, you will get a compilation error if your code attempts to change the value once set.

Using Local Variable Type Inference

Java is a statically typed language, which means each variable and each parameter has a defined type. As Java has provided the ability to create more complex types, especially related to collections, the Java syntax for variable types has gotten more and more complex. To help with this, Java version 10 introduced the concept of local variable type inference.

With this, you can declare a variable of the `var` type. So long as it is fully clear what type the variable really should be, the Java compiler will take care of the details for you. Here's an example:

```
var s = new String("Hello");
```

This example creates a new `String` for the `s` variable. Even though s is declared with the `var` keyword, `s` really is of the `String` type. That is, this code is equivalent to the following:

```
String s = new String("Hello");
```

With just a `String` type, this doesn't save you that much typing. When you get to more complex types, though, you will really appreciate the use of the `var` keyword.

Note

Chapter 4, *Collections, Lists, and Java's Built-In APIs*, covers collections, where you will see really complex types.

`Example21.java` shows local variable type inference in action:

```
public class Example21 {
    public static void main(String[] args) {
        var s = new String("Hello");
        System.out.println("The value is " + s);
        var i = Integer.valueOf("42");
        System.out.println("The value is " + i);
    }
}
```

When you run this example, you will see the following output:

```
The value is Hello
The value is 42
```

Activity 1: Taking Input and Comparing Ranges

You are tasked with writing a program that takes a patient's blood pressure as input and then determines if that blood pressure is within the ideal range.

Blood pressure has two components, the systolic blood pressure and the diastolic blood pressure.

According to https://packt.live/2oaVsgs, the ideal systolic number is more than 90 and less than 120. 90 and below is low blood pressure. Above 120 and below 140 is called pre-high blood pressure, and 140 and over is high blood pressure.

The ideal diastolic blood pressure is between 60 and 80. 60 and below is low. Above 80 and under 90 is pre-high blood pressure, and over 90 is high blood pressure.

Component	Ideal Range
Systolic blood pressure	90-120
Diastolic blood pressure	60-80

Figure 2.4: Ideal ranges for systolic and diastolic blood pressures

For the purpose of this activity, if either number is out of the ideal range, report that as non-ideal blood pressure:

1. Write an application that takes two numbers, the systolic blood pressure and the diastolic blood pressure. Convert both inputs into `int` values.
2. Check if there is the right number of inputs at the beginning of the program. Print an error message if any inputs are missing. Exit the application in this case.
3. Compare against the ideal rates mentioned previously. Output a message describing the inputs as low, ideal, pre-high, or high blood pressure.

   To print an error message, use `System.err.println` instead of `System.out.println`.

4. Try out your program with a variety of inputs to ensure it works properly.

   You'll need to use the Terminal pane in IntelliJ to compile and run the program with command-line input. Look back at Exercises 15 and 16 for details on how to do this.

5. The blood pressure is typically reported as systolic blood pressure/diastolic blood pressure.

   Note

   The solution for this activity can be found via this link.

Summary

This chapter covered a lot of Java syntax—things you need to learn to be able to tackle the more advanced topics. You'll find yourself using these techniques in just about every Java application you write.

We started out by controlling the flow of the program using conditional statements such as `if`, `else if`, `else`, and `switch` statements. We then moved on to the different loops that can be used to perform repetitive tasks. After this, we looked at how to provide values during runtime using command-line arguments. This is one way to pass inputs to your Java applications. Every example in this chapter created a class, but we have not yet done much with these classes.

In the next chapter, you'll learn about classes, methods, and object-oriented programming, and how you can do a lot more with classes.

Object-Oriented Programming

Overview

In this chapter, we will consider the way in which Java implements **object-oriented programming** (**OOP**) concepts. For these purposes, you will first practice creating and instantiating your own classes so that you can later create methods that can handle data within them. We will then take you through how to code recursive methods, and even how to override existing methods in favor of your own. By the end of the chapter, you will be fully equipped to overload the definition of methods in order to accommodate different scenarios with different parameters to the same method or constructor, and annotate code to inform the compiler about specific actions that must be taken.

Introduction

A Java class is a template that is used to define data types. Classes are composed of objects carrying data and methods that are used to perform operations on that data. Classes can be self-contained, extend other classes with new functionalities, or implement features from other classes. In a way, classes are categories that allow us to define what kind of data can be stored within them, as well as the ways in which that data can be handled.

Classes tell the compiler how to build a certain object during runtime. Refer to the explanation of what objects are in the *Working with Objects in Java* topic.

The basic structure of a class definition looks like this:

```
class <name> {
    fields;
    methods;
}
```

Note

Class names should start with a capital letter, as in `TheClass`, `Animal`, `WordCount`, or any other string that somehow expresses the main purpose of the class. If contained in a separate file, the filename containing the source should be named like the class: `TheClass.java`, `Animal.java`, and so on.

The Anatomy of a Class

There are different software components in classes. The following example shows a class that includes some of the main ones.

Example01.java

```
1  class Computer {
2      // variables
3      double cpuSpeed;  // in GHz
4
5      // constructor
6      Computer() {
7          cpuSpeed = 0;
8      }
9
10     //methods
11     void setCpuSpeed ( double _cpuSpeed ) {
12         cpuSpeed = _cpuSpeed;
13     }
```

https://packt.live/32w1ffg

The outcome of this example is:

```
2.5

Process finished with exit code 0
```

The previous code listing shows the definition of a basic class called `Computer`, which includes variables and methods to deal with one of the properties of the class computer – in this case, `cpuSpeed`. The code shows two different classes. The first one is the blueprint for how to define objects of the `Computer` type in your programs. The second one, `Example01`, is the one that will be executed after compilation and will make an instance of the `Computer` class in the form of an object called `myPC`.

There is one more component inside the class, the constructor, which is optional, as Java includes a default constructor for all your classes. **Constructors** are used to initializing the basic properties of classes, and so are used when assigning values to variables, for instance. In our case, the operation performed by the constructor is initializing the `cpuSpeed` variable with a value of `0`:

```
// constructor
Computer() {
    cpuSpeed = 0;
}
```

It is also possible for constructors to have parameters. You could have the constructor of the class be this:

```
// constructor
Computer( double _c ) {
    cpuSpeed = _c;
}
```

In this way, you could call the constructor with:

```
Computer myPC = new Computer( 2.5 );
```

That would also require a parameter. In addition, classes can have more than one constructor. This will be explained later in the chapter.

Working with Objects in Java

Objects are to classes what variables are to data types. While classes define the structure and possible actions of a certain data type, objects are actual usable parts of the computer memory containing that data. The action of creating an object is known as making an instance of a class. In a sense, it is like making a copy of the template and then modifying it by accessing its variables or methods. Let's see this in action:

```
Computer myPC = new Computer( 2.5 );
```

`myPC` is the actual object. We would say that `myPC` is an object of the `class Computer` in colloquial terms.

The different fields and methods inside the class can be accessed by typing the name of the object followed by a period and the name of the variable or method you want to address. Making any changes to the variables or calling the methods will take effect only within the scope of that object. If you had more objects of the same class in your program, each one of them would have a piece of memory of its own. An example of how to address a method is as follows:

```
myPC.setCpuSpeed( 2.5 );
```

An example of how to address a variable, on the other hand, would be the following assignment:

```
myPC.cpuSpeed = 2.5;
```

Because of the way the `Computer` class has been defined, the last two code listings have the exact same effect. The whole class has been defined—by default—as `public`, which means that all the methods, variables, and objects from the class are available to be called with the mechanism described previously. It could be necessary to prevent users

from directly interacting with the variables within the class and only allow their modification through certain methods.

The different components within a class can be defined as `public` or `private`. The former will make the component available to be used as shown so far, while the latter will hinder the ability of other developers to access that part. The following example shows how to make the `cpuSpeed` variable `private`:

Example02.java

```
1  class Computer {
2      // variables
3      private double cpuSpeed;  // in GHz
4
5      // constructor
6      Computer() {
7          cpuSpeed = 0;
8      }
9
10     // methods
11     void setCpuSpeed ( double _cpuSpeed ) {
12         cpuSpeed = _cpuSpeed;
13     }
14
15     double getCpuSpeed () {
16         return cpuSpeed;
17     }
18 }
```

https://packt.live/2pBgWTS

The result of this code listing is the same as before:

```
2.5

Process finished with exit code 0
```

If you tried to access the `cpuSpeed` variable directly from the `Example02` class, the IDE will display a compilation error. The following example shows such a case. Try it out to see how the debugger informs you when you try to access a `private` variable:

Example03.java

```
20 public class Example03 {
21     public static void main(String[] args) {
22         Computer myPC = new Computer();
23         myPC.setCpuSpeed( 2.5 );
24         System.out.println( myPC.cpuSpeed );
25     }
26 }
```

https://packt.live/2pvLu9Q

The result of this program is:

```
Example03.java:23: error: cpuSpeed has private access in Computer
        System.out.println( myPC.cpuSpeed );
1 error

Process finished with exit code 1.
```

Since we are trying to access a private element from outside a class, the compiler shows an error in the `Computer` class.

Checking the Precedence of a Class with instanceof

You can check whether an object is an instance of a specific class. This can be convenient for things such as error checking, handling data in different ways depending on its precedence, and more. The following example shows the `checkNumber` method, which can discriminate between different types of variables and will print different messages based on that:

```
public class Example04 {
    public static void checkNumber(Number val) {
        if( val instanceof Integer )
            System.out.println("it is an Integer");
        if( val instanceof Double )
            System.out.println("it is a Double");
    }
    public static void main(String[] args) {
        int val1 = 3;
        double val2 = 2.7;
        checkNumber( val1 );
        checkNumber( val2 );
    }
}
```

The outcome of the previous example is:

```
it is an Integer
```

```
it is a Double

Process finished with exit code 0
```

Exercise 1: Creating the WordTool Class

The goal of this exercise is to create a piece of reusable code, a class, that can be inserted in other programs, and that can be used to perform various statistical analyses on text. We could be interested, for example, in getting to know the frequency of a certain word or letter in the text, how many words we have used so far, and more.

`WordTool` is a class that will help you to perform a series of operations on a piece of text, including counting the number of words, looking at the frequency of letters, and searching for the occurrence of a specific string.

1. Open IntelliJ and click on the `File` | `New` | `Project` menu options:

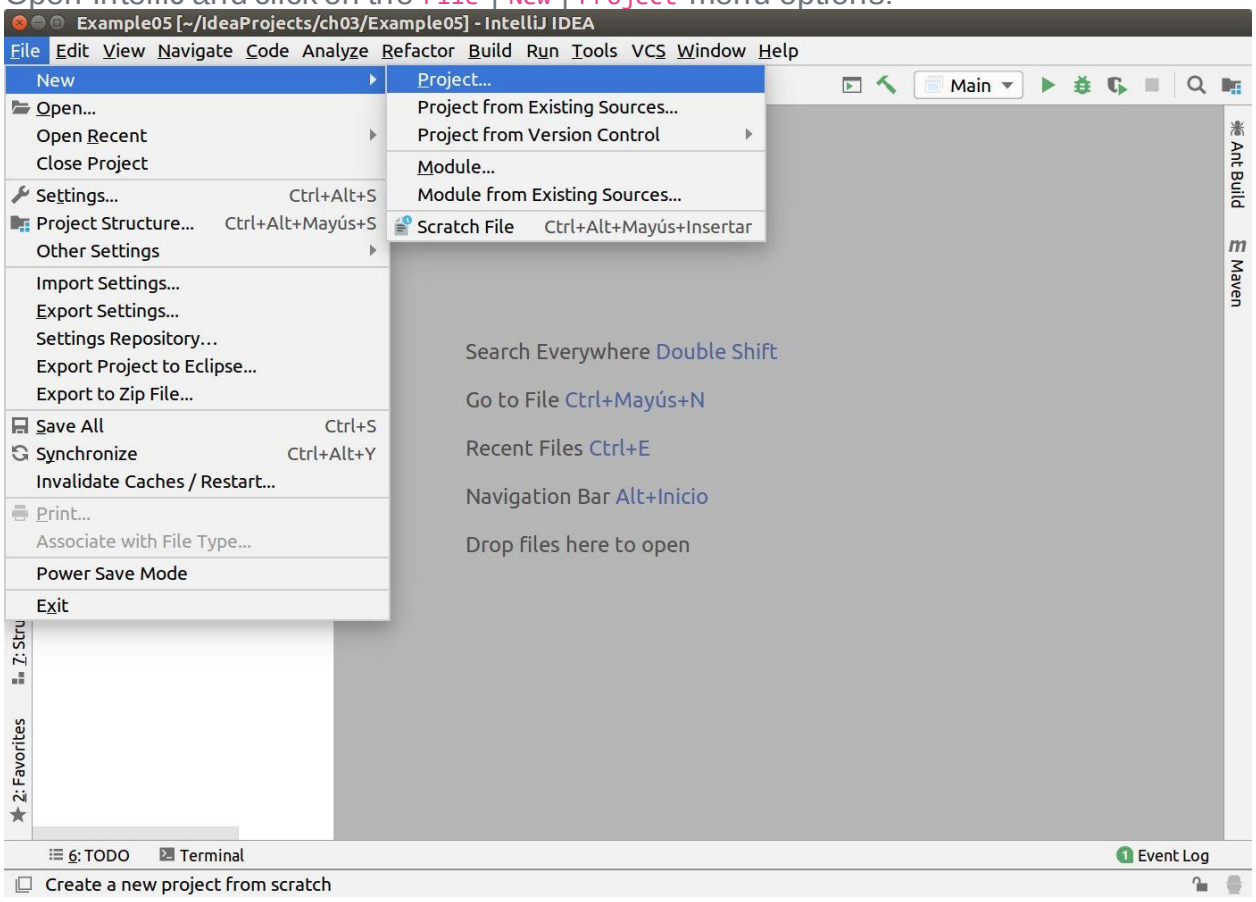


Figure 3.1: Creating a new project

2. A new interface unfolds. The default options are meant for creating a Java program. You just need to click `Next`:

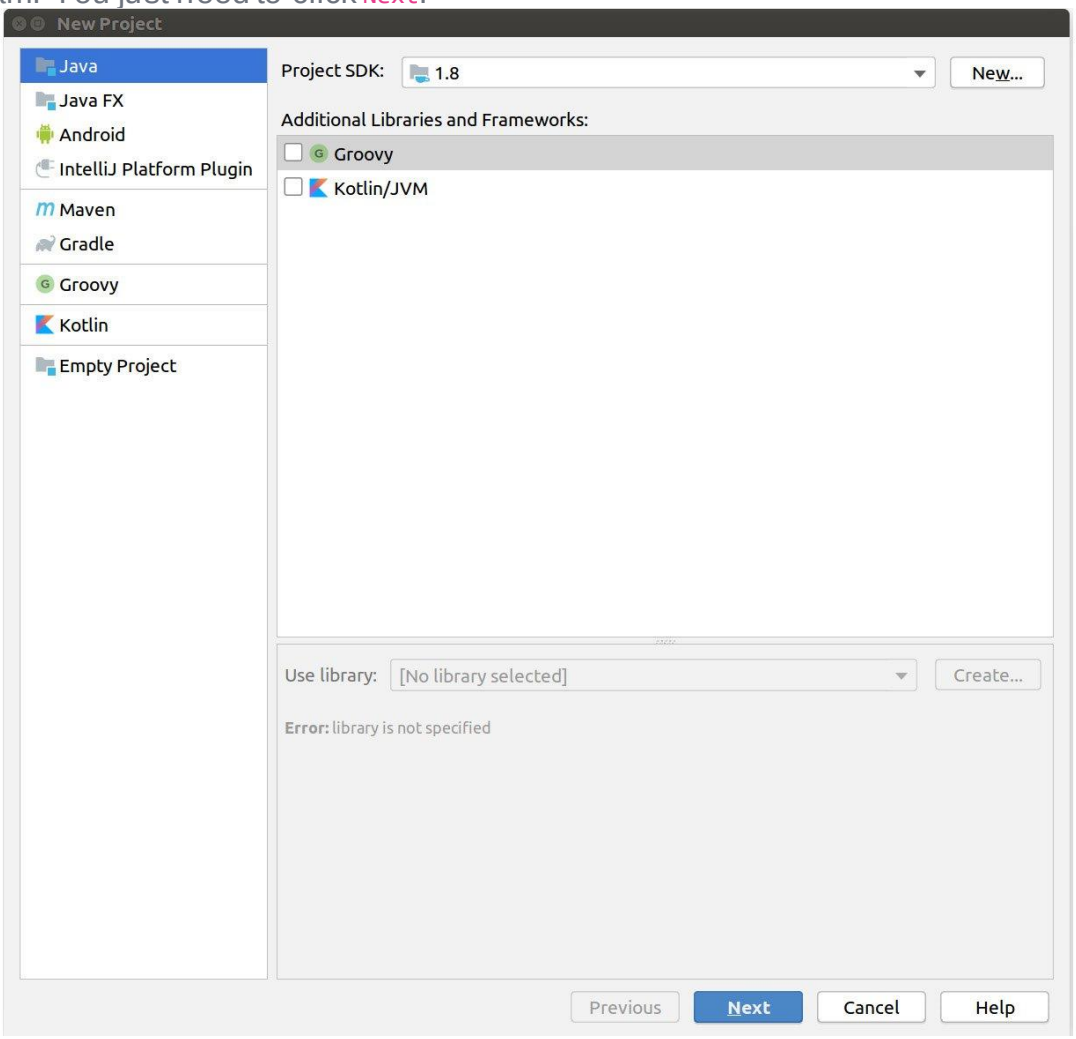


Figure 3.2: The New Project dialog box

3. Check the box to create the project from a template. Select the template for the `Command Line App`. Click `Next`:

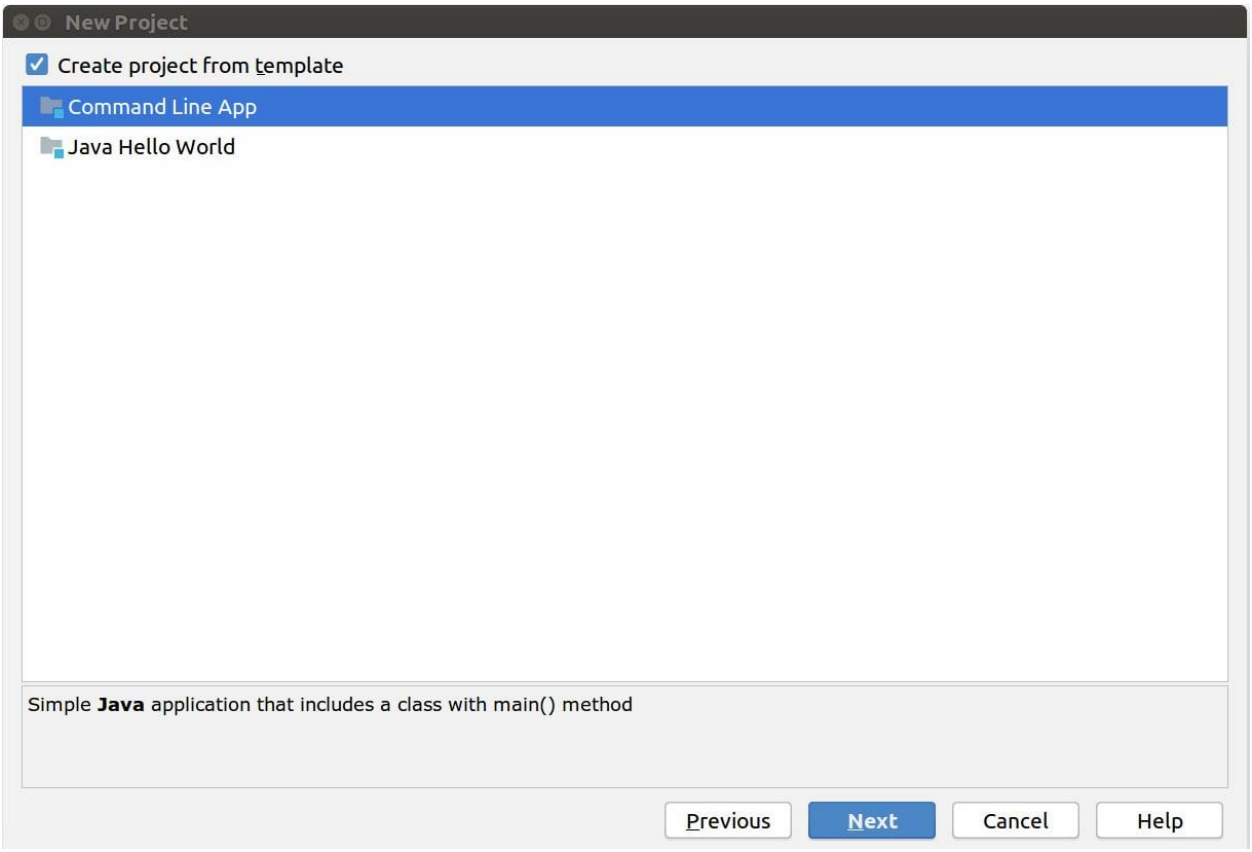


Figure 3.3: Creating a project from template

4. Name the project `WordTool`. Click `Finish`:

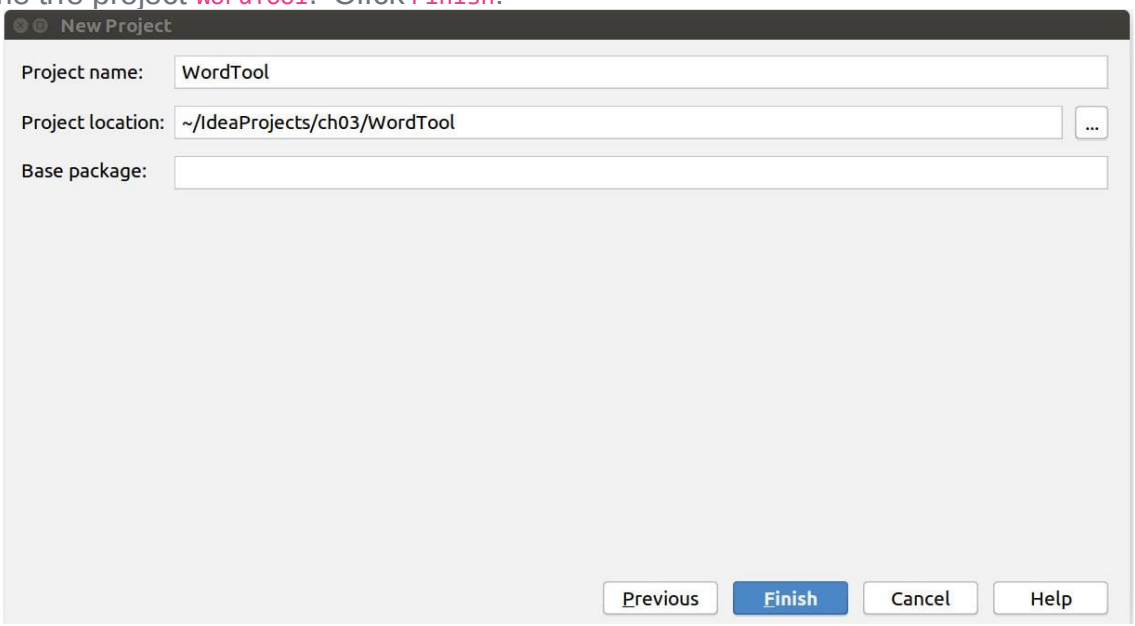


Figure 3.4: Adding the Project name

5. By default, the template calls your basic class `Main`. Let's change that to `WordTool`. First, navigate within the new project to the `Main.java` file; it is displayed as the `main` entry in the list:

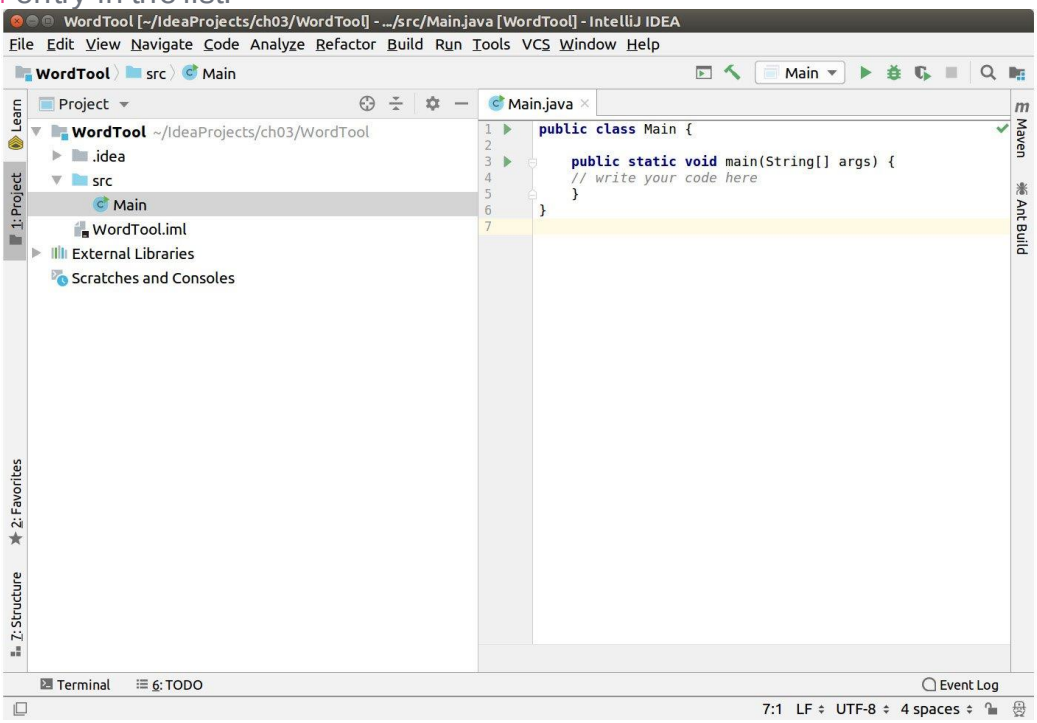


Figure 3.5: A template Java program

6. Right-click on the `Main` entry and, in the drop-down list, select the `Refactor` option. Within that, select `Rename...`:

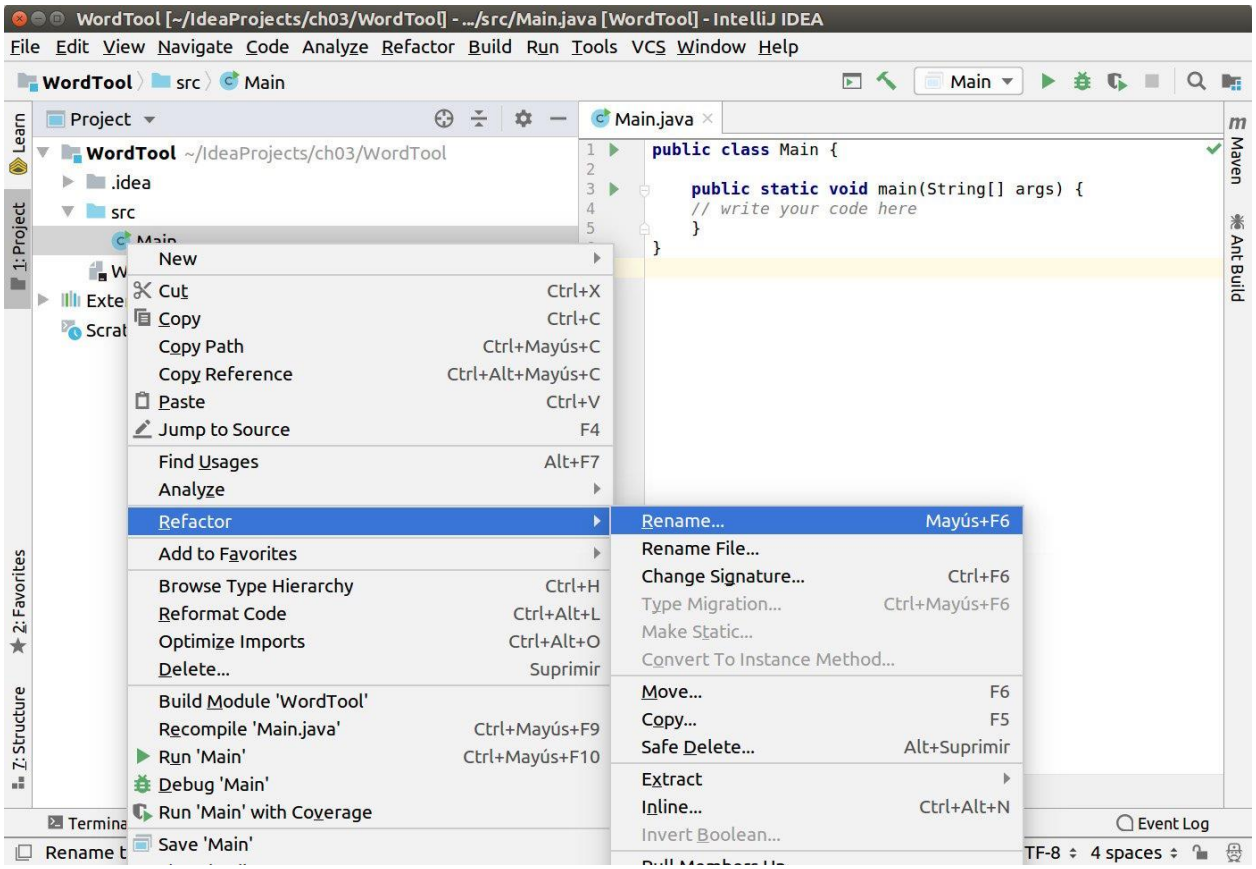


Figure 3.6: Refactoring the Java class

7. A dialog window pops up. Write in it the name of the class, `WordTool`. The checkboxes allow you to choose which parts of the code will be refactored to fit the new name of the class:

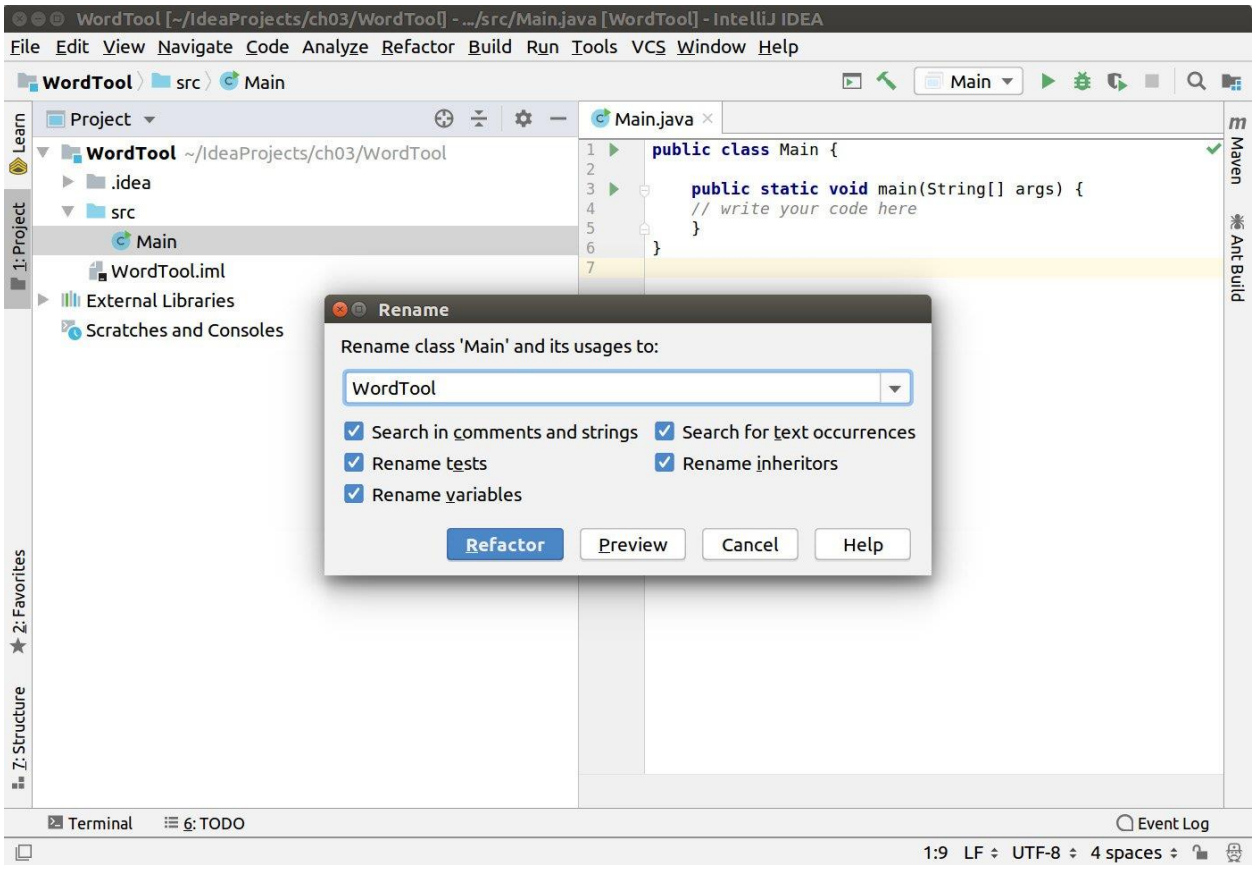


Figure 3.7: Renaming the class in IntelliJ

8. You will see that the class is now called `WordTool` and the file is `WordTool.java`:

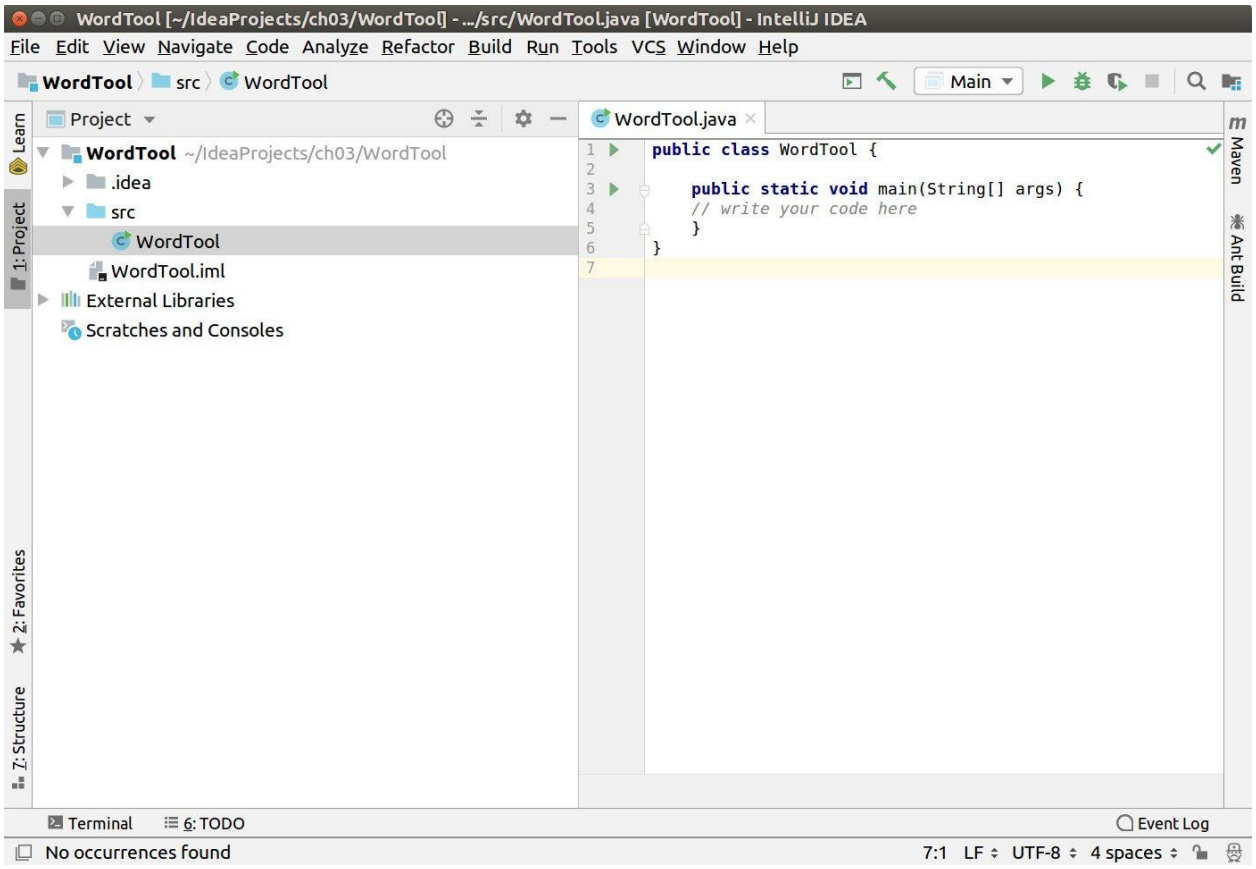


Figure 3.8: WordTool

9. Inside the `WordTool` class, create the default constructor for the class; it will be empty, in this case:

```
WordTool() {};
```

10. Let's include a method to count the number of words within a string variable. To avoid any potential errors or exceptions when operating the string, we need to perform a test before performing the count. By comparing the string to `null` or checking whether it is empty we will know if the sum of characters should be zero before even starting to count. On the other hand, the counting is performed using the `split()` method for strings. In this case, we divide the string by whitespace characters using the regular expression `\\s+`.

```
11.      public int wordCount ( String s ) {
12.          int count = 0;    // variable to count words
13.          // if the entry is empty or is null, count is zero
```

```
14.           // therefore we evaluate it only otherwise
15.           if ( !(s == null || s.isEmpty()) ) {
16.               // use the split method from the String class to
17.               // separate the words having the whitespace as
    separator
18.               String[] w = s.split("\\s+");
19.               count = w.length;
20.           }
21.           return count;
      }
```

22. Add a method to count the number of letters in a string, and add the ability to count both with and without whitespace characters:

```
23.       public int symbolCount ( String s, boolean withSpaces ) {
24.           int count = 0;  // variable to count symbols
25.           // if the entry is empty or is null, count is zero
26.           // therefore we evaluate it only otherwise
27.           if ( !(s == null || s.isEmpty()) ) {
28.               if (withSpaces) {
29.                   // with whitespaces return the full length
30.                   count = s.length();
31.               } else {
32.                   // without whitespaces, eliminate whitespaces
33.                   // and get the length on the fly
34.                   count = s.replace(" ", "").length();
35.               }
```

```
            }
            return count;
        }
```

This method takes two parameters: a string and a boolean. The first one is the data we will be reading, while the second determines whether we will count the whitespaces as symbols. It is a common technique, when creating a word counting tool, to account for all symbols with and without the whitespaces.

38. In the `main` class, create an object of the `WordTool` class and add a `String` variable containing some text of your choice:

```
    WordTool wt = new WordTool();
    String text = "The river carried the memories from her childhood.";
```

40. Add code inside the main method to print out the calculations made by `WordTool`:

```
    System.out.println( "Analyzing the text: \n" + text );
    System.out.println( "Total words: " + wt.wordCount(text) );
    System.out.println( "Total symbols (w. spaces): " +
    wt.symbolCount(text, true) );
    System.out.println( "Total symbols (wo. spaces): " +
    wt.symbolCount(text, false) );
```

44. Run the program; the outcome should be as follows:

```
    Analyzing the text:
    The river carried the memories from her childhood.
    Total words: 8
    Total symbols (w. spaces): 50
    Total symbols (wo. spaces): 43
    Process finished with exit code 0
```

Note

You can use the trick presented in this exercise to create classes for all the examples in this book, just by using the template and refactoring them to have the example name. After that, you will just need to copy the code in a fresh project.

Activity 1: Adding the Frequency-of-Symbol Calculation to WordTool

Add a method to the previously created `WordTool` class to calculate the frequency of a certain symbol. To do so, perform the following steps:

1. Add a method to count the number of words in a string.
2. Add a method to count the number of letters in a string, and add the possibility of separating the case of having whitespaces or not.
3. In the `main` class, create an object of the `WordTool` class and add a string variable containing a line of text of your choice.
4. Add code inside the main method to print out the calculations made by WordTool.

The expected outcome of this activity is as follows:

```
Analyzing the text:
The river carried the memories from her childhood.
Total words: 8
Total symbols (w. spaces): 50
Total symbols (wo. spaces): 43
Total amount of e: 7
Process finished with exit code 0
```

Note

The solution for this activity can be found via this link.

Inheritance in Java

Inheritance is a key principle of object-oriented programming. It entails the transfer of the existing structure of one class, including its constructor, variables, and methods, to a different class. The new class is called the child class (or subclass), while the one it's inheriting from is called the parent class (or superclass). We say that the child class extends the parent one. The child class is said to extend the parent class in the sense that it not only inherits whatever structures are defined by the parent, but it also creates new structures. The following example shows a parent class and how the child class extends it by adding a new method to it. We will take the `Computer` class we defined earlier as a parent and create a new class called `Tablet`, which is a type of computer.

Example05.java

```
20 class Tablet extends Computer {
21     // variables
22     private double screenSize;  // in inches
23
24     // methods
25     void setScreenSize ( double _screenSize ) {
26         screenSize = _screenSize;
27     }
28
29     double getScreenSize () {
30         return screenSize;
31     }
32 }
33
34 public class Example05 {
35     public static void main(String[] args) {
36         Tablet myTab = new Tablet();
```

```
37          myTab.setCpuSpeed( 2.5 );
38          myTab.setScreenSize( 10 );
39          System.out.println( myTab.getCpuSpeed() );
40          System.out.println( myTab.getScreenSize() );
41      }
42 }
```

https://packt.live/2o3NaqE

Notice how the definition of the `Tablet` class does not include any methods called `setCpuSpeed()` or `getCpuSpeed()`; however, when calling them, not only does the program not give any errors, but the commands are also successfully launched.

This is because the definition of the `Tablet` class extends the `Computer` class, thus inheriting all its internal objects, variables, and methods. When creating an object of the `Tablet` class, such as `myTab`, the JVM reserves space in memory for a `cpuSpeed` variable and the setter and getter methods that go with it.

Overriding and Hiding Methods

When extending a class, it is possible to redefine some of the methods that are part of it. Overriding means to rewrite something's functionality. This is done by making a new declaration of the method with the same name and properties of the method from the original class. This is demonstrated in the following example. Note that we're continuing, for the sake of clarity, with `Computer` and `Tablet`, but they have been cleaned up so as not to make the example programs too long.

```
class Computer {
    public void whatIsIt() {
        System.out.println( "it is a PC");
    }
```

```
}
class Tablet extends Computer {
    public void whatIsIt() {
        System.out.println( "it is a tablet");
    }
}
class Example06 {
    public static void main(String[] args) {
        Tablet myTab = new Tablet();
        myTab.whatIsIt();
    }
}
```

Since `Tablet` extends `Computer`, you could modify the main class in the program to be as follows:

```
class Example06 {
    public static void main(String[] args) {
        Computer myTab = new Tablet();
        myTab.whatIsIt();
    }
}
```

Technically, tablets are computers, which means that you can create an object of the `Tablet` class by defining it as `Computer` in the first place. The result for both cases will be the same:

```
it is a tablet
```

```
Process finished with exit code 0
```

The result is the same for both classes because both the child and parent classes include a non-static method called `whatIsIt()`. When calling the method, the overriding one will have priority. This is done by the JVM at runtime. This principle is what we call runtime polymorphism. There can be multiple definitions of the same method, and which definition will be executed is decided during the execution of the program.

But what would happen if the method you called was static? This could be a design decision taken by the developer who is creating the class you are extending and therefore is a situation out of your control. In this case, it is not possible to override the method. The child class can, however, hide the method defined by the parent using the same mechanism. The next code listing demonstrates this.

```
class Computer {
    public static void whatIsIt() {
        System.out.println( "it is a PC");
    }
}
class Tablet extends Computer {
    public static void whatIsIt() {
        System.out.println( "it is a tablet");
    }
}
class Example07 {
    public static void main(String[] args) {
        Computer myTab = new Tablet();
        myTab.whatIsIt();
    }
```

```
}
```

The outcome of this example is:

```
it is a PC

Process finished with exit code 0
```

The decision of what method should be used with static methods is not taken at runtime but during compilation, and this ensures that the method from the parent class is the one being called. This action is called **hiding instead of overriding**. It is still possible to call the method in the `Tablet` class. To do so, you should modify the code in the `main` class to the following:

```
class Example07 {
    public static void main(String[] args) {
        Computer myTab = new Tablet();
        Tablet.whatIsIt();
    }
}
```

Note how we clearly specify the actual class you call for this. The result of the modified example is:

```
it is a tablet

Process finished with exit code 0
```

Avoiding Overriding: Final Classes and Methods

If you want to stop other developers from overriding parts of your class, you can declare the methods you want to protect as `final`. An example of this could be a class that deals with temperature. The method that converts from Celsius into Fahrenheit is final, as it makes no sense to override such a method.

```
class Temperature {
    public double t = 25;
    public double getCelsius() {
        return t;
    }
    final public double getFahrenheit() {
        return t * 9/5 + 32;
    }
}
class Example08 {
    public static void main(String[] args) {
        Temperature temp = new Temperature();
        System.out.println( temp.getCelsius() );
        System.out.println( temp.getFahrenheit() );
    }
}
```

This program will give this result:

```
25.0
77.0
Process finished with exit code 0
```

Note

Alternatively, you can declare a whole class `final`. A `final` class cannot be extended. An example of such a class is `String`. You could ask whether it defeats the purpose of object-oriented programming to have a class that cannot be extended. But there are

some classes that are so fundamental to the programming language, such as `String`, that they are better kept as they are.

Overloading Methods and Constructors

One very interesting property of Java is how it allows you to define methods that have the same conceptual functionality as each other by using the same name but changing either the type or number of parameters. Let's see how this could work.

```
class Age {
    public double a = 0;
    public void setAge ( double _a ) {
        a = _a;
    }
    public void setAge ( int year, int month ) {
        a = year + (double) month / 12;
    }
    public double getAge () {
        return a;
    }
}
class Example09 {
    public static void main(String[] args) {
        Age age = new Age();
        age.setAge(12.5);
        System.out.println(age.getAge());
        age.setAge(9, 3);
```

```
        System.out.println(age.getAge());
    }
}
```

Note

Look at the highlighted portion in the preceding code. As we are taking the integer parameter `month` and dividing it by a number, the result of the operation will be a double. To avoid possible errors, you need to convert the integer into a floating comma number. This process, called casting, is done by adding the new type between brackets in front of the object, variable, or operation we want to convert.

The result of this example is:

```
12.5
9.25
Process finished with exit code 0
```

This shows that both methods modify the `a` variable in the `Age` class by taking different sets of parameters. This same mechanism for having conceptually equivalent results from different blocks of code can be used for the constructors of a class, as shown in the following example.

```
class Age {
    public double a = 0;
    Age ( double _a ) {
        a = _a;
    }
    Age ( int year, int month ) {
        a = year + (double) month / 12;
    }
```

```
    public double getAge () {
        return a;
    }
}
class Example10 {
    public static void main(String[] args) {
        Age age1 = new Age(12.5);
        Age age2 = new Age(9, 3);
        System.out.println(age1.getAge());
        System.out.println(age2.getAge());
    }
}
```

In this case, as a way to show the functionality, instead of instantiating a single object and calling the different methods to modify its variables, we had to create two different objects, `age1` and `age2`, with one or two parameters, as those are the possible options offered by the constructors available in the `Age` class.

Recursion

Programming languages allow the usage of certain mechanisms to simplify solving a problem. Recursion is one of those mechanisms. It is the ability of a method to call itself. When properly designed, a recursive method can simplify the way a solution to a certain problem is expressed using code.

Classic examples in recursion include the computation of the factorial of a number or the sorting of an array of numbers. For the sake of simplicity, we are going to look at the first case: finding the factorial of a number.

```
class Example11 {
```

```
    public static long fact ( int n ) {
        if ( n == 0 ) return 1;
        return n * fact ( n – 1 );
    }
    public static void main(String[] args) {
        int input = Integer.parseInt(args[0]);
        long factorial = fact ( input );
        System.out.println(factorial);
    }
}
```

To run this code, you will need to go to the terminal and call the example from there with `java Example11 m`, where `m` is the integer whose factorial will be calculated. Depending on where you created the project on your computer, it could look like this (note that we have shortened the path to the example to keep it clean):

```
usr@localhost:~/IdeaProjects/chapter03/[...]production/Example11$ java
Example11 5
120
```

Or, it could look like this:

```
usr@localhost:~/IdeaProjects/chapter03/[...]production/Example11$ java
Example11 3
6
```

The result of the call is the factorial: `120` is the factorial of `5`, and `6` is the factorial of `3`.

While it might not seem so intuitive at first sight, the `fact` method calls itself in the return line. Let's take a closer look at this:

```
public static long fact ( int n ) {
    if ( n == 0 ) return 1;
    return n * fact ( n – 1 );
}
```

There are a couple of conditions that you need to meet when designing a functional recursive method. Otherwise, the recursive method will not converge to anything:

1. There needs to be a base condition. This means you need something that will stop the recursion from happening. In the case of the `fact` method, the base condition is `n` being equal to 0:
   ```
   if ( n == 0 ) return 1;
   ```
2. There needs to be a way to computationally reach the base condition after a certain number of steps. In our case, every time we call fact, we do it with a parameter that is one unit smaller than the parameter of the current call to the method:
   ```
   return n * fact ( n – 1 );
   ```

Annotations

Annotations are a special type of metadata that can be added to your code to inform the compiler about relevant aspects of it. Annotations can be used during the declaration of classes, fields, methods, variables, and parameters. One interesting aspect of annotations is that they remain visible inside classes, indicating whether a method is an override to a different one from a parent class, for example.

Annotations are declared using the `@` symbol followed by the annotation's name. There are some built-in annotations, but it is also possible to declare your own. At this point, it is important to focus on some of the built-in ones, as it will help you to understand some of the concepts presented so far in this chapter

The most relevant built-in annotations are `@Override`, `@Deprecated`, and `@SuppressWarnings`. These three commands inform the compiler about different aspects of the code or the process of producing it.

`@Override` is used to indicate that a method defined in a child class is an override of another one in a parent class. It will check whether the parent class has a method named the same as the one in the child class and will provoke a compilation error if it doesn't exist. The use of this annotation is displayed in the following example, which builds on the code we showcased earlier in the chapter about the `Tablet` class extending the `Computer` class.

```
class Computer {
    public void whatIsIt() {
        System.out.println( "it is a PC");
    }
}
class Tablet extends Computer {
    @Override
    public void whatIsIt() {
        System.out.println( "it is a tablet");
    }
}
class Example12 {
    public static void main(String[] args) {
        Tablet myTab = new Tablet();
        myTab.whatIsIt();
    }
}
```

`@Deprecated` indicates that the method is about to become obsolete. This typically means that it will be removed in a future version of the class. As Java is a living language, it is common for core classes to be revised and new methods to be produced, and for the functionality of others to cease being relevant and get deprecated. The following

example revisits the previous code listing, if the maintainer of the `Computer` class has decided to rename the `whatIsIt()` method `getDeviceType()`.

Example13.java

```
1  class Computer {
2      @Deprecated
3      public void whatIsIt() {
4          System.out.println( "it is a PC");
5      }
6
7      public void getDeviceType() {
8          System.out.println( "it is a PC");
9      }
10 }
11
12 class Tablet extends Computer {
13     @Override
14     public void whatIsIt() {
15         System.out.println( "it is a tablet");
16     }
17 }
```

https://packt.live/35NGCgG

Calling the compilation of the previous example will issue a warning about the fact that the `whatIsIt()` method will soon be no longer used. This should help developers plan their programs, as they'll know that some methods may disappear in the future:

```
Warning:(13, 17) java: whatIsIt() in Computer has been deprecated
```

`@SuppressWarnings` makes the compiler hide the possible warnings that will be defined in the annotation's parameters. It should be mentioned that annotations can have parameters such as `overrides`, `deprecation`, `divzero`, and `all`. There are more types of warnings that can be hidden, but it is too early to introduce them. While we are not going to go deeper into this concept at this point, you can see an example of this in the following code listing.

Example14.java

```
1  class Computer {
2      @Deprecated
3      public void whatIsIt() {
4          System.out.println( "it is a PC");
5      }
6
7      public void getDeviceType() {
8          System.out.println( "it is a PC");
9      }
10 }
11
12 @SuppressWarnings("deprecation")
13 class Tablet extends Computer {
14     @Override
15     public void whatIsIt() {
16         System.out.println( "it is a tablet");
17     }
```

```
18 }
```

https://packt.live/33GKnTt

When calling the compilation of the latest example, you will see a difference in comparison to the previous one, as the compilation of this one will not produce any warnings regarding the deprecation of the `whatIsIt()` method.

Note

You should be careful when using `@SuppressWarnings` as it can hide risks derived from potential malfunctions of your code. Especially avoid using `@SuppressWarnings("all")`, as it could mask warnings that could be producing runtime errors in other parts of your code.

Interfaces

Interfaces are reference types in Java. As such, they define the skeleton of classes and objects but without including the actual functionality of methods. Classes implement interfaces but do not extend them. Let's look at an example of a simple interface, further developing the idea of building classes to represent different types of computers.

```
interface Computer {
    public String getDeviceType();
    public String getSpeed();
}
class Tablet implements Computer {
    public String getDeviceType() {
        return "it is a tablet";
    }
    public String getSpeed() {
```

```
        return "1GHz";
    }
}
class Example15 {
    public static void main(String[] args) {
        Tablet myTab = new Tablet();
        System.out.println( myTab.getDeviceType() );
        System.out.println( myTab.getSpeed() );
    }
}
```

As you might have guessed, the output for this example is:

```
it is a tablet
1GHz
Process finished with exit code 0
```

Some relevant notes on interfaces follow:

- Interfaces can extend other interfaces.
- Unlike classes, which can only extend one class at a time, interfaces can extend multiple interfaces at once. You do so by adding the different interfaces separated by commas.
- Interfaces have no constructors.

Inner Classes

Classes, as we have seen so far, cannot be hidden to other parts of the program. In code terms, they cannot be made private. To offer this kind of security mechanism, Java

developed so-called **inner classes**. This type of class is declared nested within other classes. A quick example of this follows:

Example16.java

```
1  class Container {
2      // inner class
3      private class Continent {
4      public void print() {
5              System.out.println("This is an inner class");
6          }
7      }
8
9      // method to give access to the private inner class' method
10     void printContinent() {
11         Continent continent = new Continent();
12         continent.print();
13     }
14 }
```

https://packt.live/2P2vc30

The result of the previous example is:

```
This is an inner class
Process finished with exit code 0
```

The previous example is a case of a non-static inner class. There are two more: method-local inner classes (these are defined inside a method) and anonymous classes. There is no big difference in how method-local classes are declared in

comparison to what you've seen so far. A method-local inner class's main characteristic is that it is defined only for the scope of that method; it cannot be called by other parts of the program.

When it comes to anonymous inner classes, they make for an interesting case that deserves to be studied. The reason for their existence is to make code more concise. With anonymous classes, you declare and instantiate the class at the same time. This means that for such a class, only one object is created. Anonymous classes are typically created by extending existing classes or interfaces. Let's look at an example defining one of these specific types of anonymous classes:

```
class Container {
    int c = 17;
    public void print() {
        System.out.println("This is an outer class");
    }
}
class Example17 {
    public static void main(String[] args) {
        // inner class
        Container container = new Container() {
            @Override
            public void print() {
                System.out.println("This is an inner class");
            }
        };
        container.print();
        System.out.println( container.c );
```

```
    }
}
```

This example shows how an anonymous class can be created in an ad hoc way to override a single method from the original class. This is one of the many possible applications of this type of inner class. The output of this program is:

```
This is an inner class
17

Process finished with exit code 0
```

Documenting with JavaDoc

Javadoc is a tool that comes with the JDK that can be used to generate documentation of classes directly from properly commented code. It requires the use of a specific type of commenting that is different from the ones seen in *Chapter 1*, *Getting Started*. There, we saw that comments can be added to code using either `//` or `/*` or `*/`. JavaDoc uses a specific type of marking to detect what comments were intentionally made for documentation purposes. Javadoc comments are contained within `/**` and `*/`. A simple example follows.

Example18.java

```
1  /**
2   * Anonymous class example
3   * This example shows the declaration of an inner class extending
4   * an existing class and overriding a method. It can be used as a
5   * technique to modify an existing method for something more suitable
6   * to our purpose.
7   *
```

```
8   * @author Joe Smith
9   * @version 0.1
10  * @since 20190305
11  */
```

https://packt.live/2J5u4aT

Note

If you are going to generate documentation from a class, you need to make sure the class is public, otherwise, the JavaDoc generator will complain about the fact that it makes no sense to document classes that aren't public.

The new comments include information about the program itself. It is good practice to explain, in some detail, what the program does. Sometimes, it may be convenient to even add blocks of code. In order to support that extra information, there are tags that allow the addition of specific features or metadata to the documentation. `@author`, `@version`, and `@since` are examples of such metadata – they determine who made the code, the version of the code, and when it was first created, respectively. There is a long list of possible tags that you can use; visit https://packt.live/2J2Px4n for more information.

JavaDoc renders the documentation as one or more HTML files. Therefore, it is possible to also add HTML markup to help messages. You could change the documentation part of the previous example as follows:

```
/**
 * <H1>Anonymous class example</H1>
 * This example shows the declaration of an <b>inner class</b> extending
 * an existing class and overriding a method. It can be used as a
 * technique to modify an existing method for something more suitable
 * to our purpose.
```

```
 *
 * @author Joe Smith
 * @version 0.1
 * @since 20190305
 */
```

Finally, you can create the documentation file by selecting `Tools` | `Generate JavaDoc` from the menu:

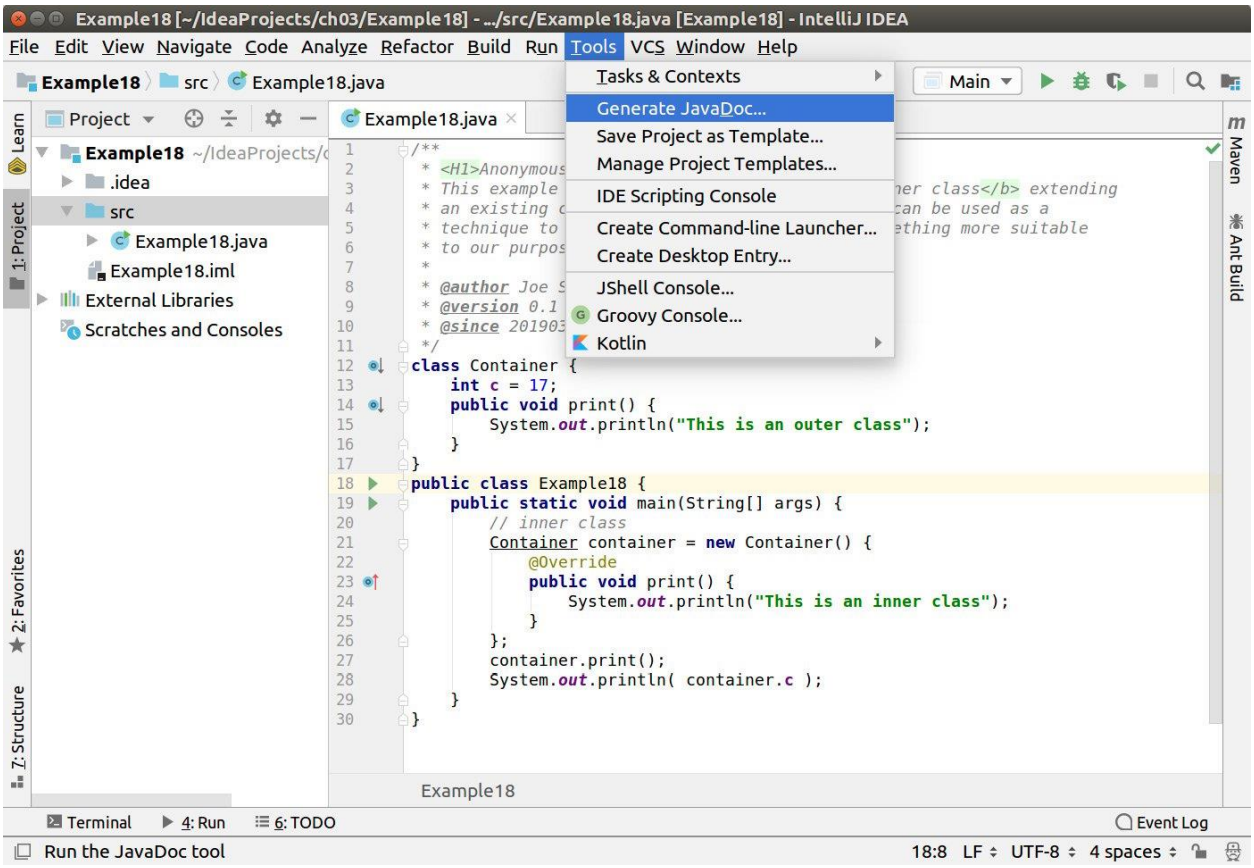


Figure 3.9: Generating JavaDoc

The JavaDoc generation dialog box will pop up and give you some options. Make sure that you insert the folder where you want the documentation file to be stored (`/tmp` in the example) and tick the checkboxes for the `@author` and the `@version`:

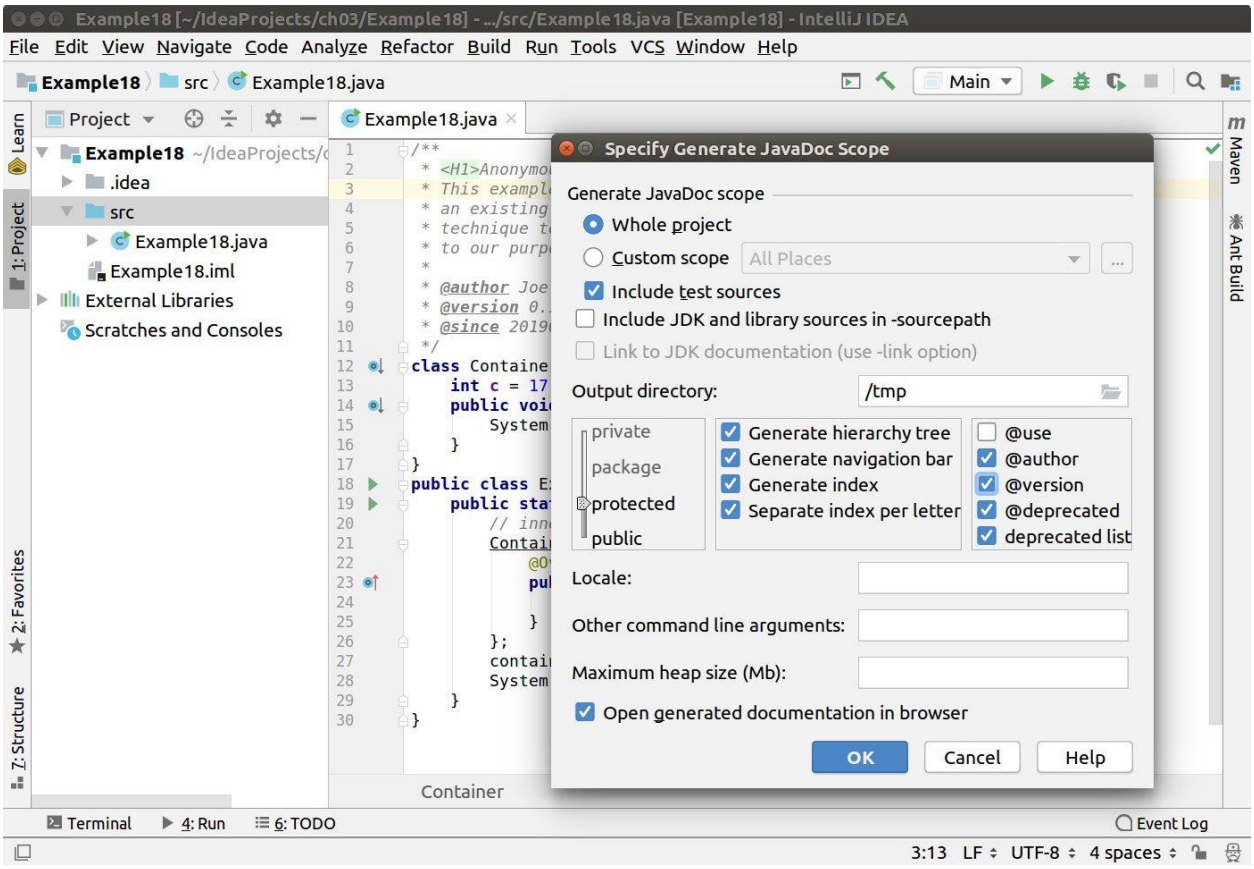


Figure 3.10: Specifying the scope for the JavaDoc

This will generate an HTML file that is formatted in the same way that official Java documentation is:

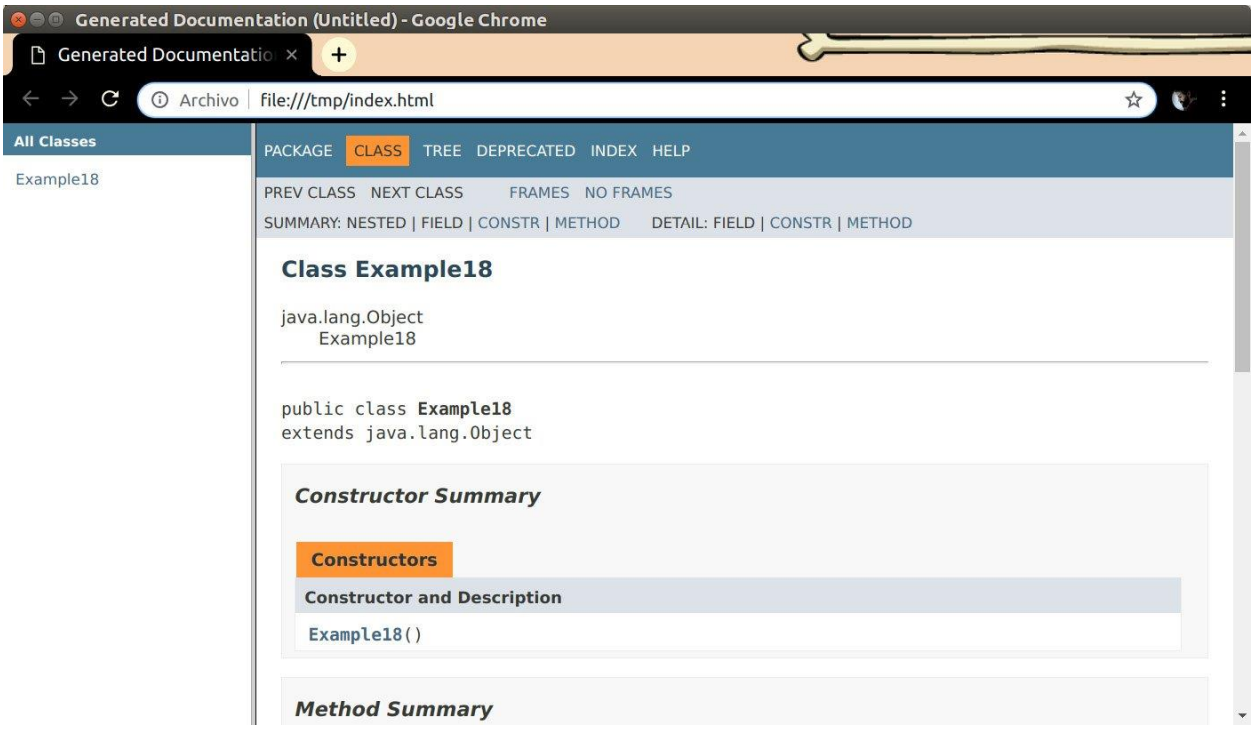


Figure 3.11: The generated JavaDoc

Activity 2: Adding Documentation to WordTool

Create documentation for the class created in *Exercise 1*, *Creating the WordTool Class*.

1. Make sure you document each one of the examples and add enough metadata for people to know how to handle the different methods.
2. Export the resulting documentation file.

   Note

   The solution for this activity can be found via this link.

Summary

This chapter introduced you to the core of object-oriented programming—the creation of classes and those operations which can be performed with them, such as extending them, using them to override parts of the code, or creating local instances.

The examples provided here showed you the importance of creating classes to better structure your code and improve it economically. If there are several classes within a specific context, it is very likely that they will have common characteristics that could be described in a parent class or even an interface.

A part of the chapter was dedicated to operations done with the compiler. As a developer, you may want to inform others when certain parts of your code will be deprecated, or whether a method from a specific class has been overridden. Annotating code is a good technique for maintaining communication with others. You also saw how to turn off possible warnings coming from annotations that occurred during development.

Finally, we discussed the process of documentation. This is relevant when sharing code or passing it over to other people. In the next chapter, we will take a look at Java collections framework which will simplify your dealings with complex data structures.

Collections, Lists and Java's Built-In APIs

Overview

This chapter introduces you to the powerful Java collections framework, which is used to store, sort, and filter data. It will first take you through the structure of the built-in Collections **Application Programming Interface** (**API**), the Java collections framework, which will simplify your dealings with complex data structures and allow you to use and create APIs with minimal effort. Through this framework, you will examine the relationship between lists and arrays, and learn to populate lists from arrays. Finally, in this chapter's final activity, you will create and complete a program in which you will be asked to perform standard operations on data stored in sets, lists, and maps in preparation for future chapters.

Introduction

Java comes with a built-in Collections API, allowing you to manipulate data structures with very little effort. A collection is an object that contains multiple elements. Collections are used to store, share, process, and communicate aggregated data. We call this system the **Java collections framework**.

As part of this framework, there are different components that are used to optimize our interaction with the actual data:

- **Interfaces**: Abstract data types that represent collections
- **Implementations**: Specific implementations of the collection interfaces
- **Algorithms**: Polymorphic methods used to process the data within a collection for operations such as sorting and searching

> Note
>
> Other programming languages have their own collection frameworks. For example, C++ has the **Standard Template Library** (**STL**). Java boasts simplicity when it comes to its collection framework.

Using the collections framework has many benefits, including a reduction in the complexity of creating programs that deal with data structures, an increase in the performance of programs, a simplification of API creation and use, and an increase in the reuse of functioning software.

The collections framework is relevant even when handling data that can be accessed by several processes simultaneously, as this would be the case in multithreaded programming scenarios. However, it is not the intention of this chapter to deal with concurrent programming.

The Collections API comes with five main interfaces:

- `Set`: A collection that contains no duplicates
- `List`: An ordered collection or sequence, allowing for duplicates
- `Queue`: A collection that sorts data in the order of its arrival, typically handled as a **First In First Out** (**FIFO**) process
- `Deque`: Essentially a queue that allows for data insertion at both ends, meaning that it can be handled both as FIFO and **Last In First Out** (**LIFO**)
- `Map`: Relates keys—which must be unique—to values

In this chapter, we will define the main interfaces (lists, sets, and maps), and explore examples of their respective uses. The framework has even more interfaces than the ones listed previously, but the others are either just variations of those listed or are outside the scope of this chapter. Furthermore, we will look at how arrays work in much more depth than we have previously.

The definition of a simple collection—in this case, a specific type of set would be as follows:

```
Set mySet = new HashSet();
```

Note

The different available classes for sets, lists, queues, deques, and maps are named after the interfaces. The different classes present different properties, as we will see later in the chapter.

Arrays

Arrays are part of the collections framework. There are some static methods that can be used to manipulate arrays. The operations you can perform are creating, sorting, searching, comparing, streaming, and transforming arrays. You were introduced to arrays in *Chapter 2*, *Learning the Basics*, where you saw how they can be used to store data of the same type. The declaration of an array is quite straightforward. Let's see what an array of strings would look like:

```
String[] text = new String[] { "spam", "more", "buy" };
```

Running operations on an array is as easy as calling some of the methods contained in the `java.util.Arrays` package. For example, sorting the previous array would require calling the following:

```
java.util.Arrays.sort( text );
```

The methods dedicated to handling arrays include one method that could be used to print out full arrays as if they were strings. This can be very handy when debugging a program:

```
System.out.println( java.util.Arrays.toString( text ) );
```

This will print the arrays and display each element separated by commas and within square brackets, `[]`. If you executed the previous command after sorting the declared array of strings, the outcome would be:

```
[buy, more, spam]
```

As you can see, the array has been sorted in ascending alphabetical order. There is a difference between that way of printing out an array and using a `for` loop to iterate throughout an array:

```
for (int i = 0; i < text.length; i++)
```

```
    System.out.print(text[i] + " ");
```

This would give the following as the result:

```
buy more spam
```

If you want to write your code in a slightly cleaner way, you could import the whole `java.util.Arrays` API at the beginning of your program, which would allow you to call the methods by omitting the `java.util` part of the command. See the following example highlighting this technique:

```
import java.util.Arrays;

public class Example01 {

    public static void main(String[] args) {

        String[] text = new String[] { "spam", "more", "buy" };

        Arrays.sort(text);

        System.out.println(Arrays.toString(text));

        for (int i = 0; i < text.length; i++)

            System.out.print(text[i] + " ");

    }

}
```

The outcome will be:

```
[buy, more, spam]

buy more spam

Process finished with exit code 0
```

If you were to make a new array that you wanted to be filled up with the same data for all cells, there is the possibility of calling the `java.util.Arrays.fill()` method, as shown here:

```
int[] numbers = new int[5];
Arrays.fill(numbers, 0);
```

Such a command would create an array filled with zeros:

```
[0, 0, 0, 0, 0]
```

Creating arrays with prefilled data can also be done with a copy of a preexisting array. It is possible to create an array by copying part of one array, or by instantiating a larger one where the old one would just be part of it. Both methods are shown in the following example, which you can test in your editor:

```
import java.util.Arrays;
public class Example02 {
    public static void main(String[] args) {
        int[] numbers = new int[5];
        Arrays.fill(numbers, 1);
        System.out.println(Arrays.toString(numbers));
        int [] shortNumbers = Arrays.copyOfRange(numbers, 0, 2);
        System.out.println(Arrays.toString(shortNumbers));
        int [] longNumbers = Arrays.copyOf(numbers, 10);
        System.out.println(Arrays.toString(longNumbers));
    }
}
```

This example will print the `numbers`, `shortNumbers` (which is shorter), and `longNumbers` (which is longer) arrays. The newly added positions in the array will be filled with zeros. If it was an array of strings, they would be filled up with `null`. The outcome of this example is:

```
[1, 1, 1, 1, 1]
[1, 1]
[1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
Process finished with exit code 0
```

You can compare arrays by calling the `java.utils.Arrays.equals()` or `java.util.Arrays.deepEquals()` methods. The difference between them is that the latter can look through nested arrays. A simple comparison example of the former method in use follows:

```
import java.util.Arrays;
public class Example03 {
    public static void main(String[] args) {
        int[] numbers1 = new int[3];
        Arrays.fill(numbers1, 1);
        int[] numbers2 = {0, 0, 0};
        boolean comparison = Arrays.equals(numbers1, numbers2);
        System.out.println(comparison);
        int[] numbers3 = {1, 1, 1};
        comparison = Arrays.equals(numbers1, numbers3);
        System.out.println(comparison);
        int[] numbers4 = {1, 1};
        comparison = Arrays.equals(numbers1, numbers4);
        System.out.println(comparison);
    }
}
```

In this example, we create four arrays: `numbers1`, `numbers2`, `numbers3`, and `numbers4`. Only two of them are the same, containing three instances of `1`. In the example, you can see how the last three arrays are compared to the first one. You can also see how the last array differs not in content, but in size. The outcome of this code is:

```
false

true

false

Process finished with exit code 0
```

Note

Since this chapter is not looking into such a complex data structure as nested arrays, we will not show an example of `java.util.Arrays.deepEquals()`. If you're interested, you should consider checking the Java reference at https://packt.live/2MuRrNa.

Searching within arrays is done through different algorithms behind the scenes. It is obviously a lot faster to perform searches on sorted arrays than on unsorted ones. The method to be invoked to run such a search on a sorted array is `Arrays.binarySearch()`. As it has many possible parameter combinations, it is recommended to visit the official documentation for the method. The following example illustrates how it works:

```
import java.util.Arrays;

public class Example04 {

    public static void main(String[] args) {

        String[] text = {"love","is", "in", "the", "air"};

        int search = Arrays.binarySearch(text, "is");

        System.out.println(search);

    }

}
```

This code is going to search for the word `is` inside the array text. The result is:

```
-4

Process finished with exit code 0
```

This is incorrect. `binarySearch` is an optimized search algorithm within the collections framework, but it is not optimal when used with unsorted arrays. This means that `binarySearch` is mainly very useful for determining whether an object can be found within an array (by sorting it first). At the same time, we will need a different algorithm when we must search through unsorted arrays and when there are multiple occurrences of a value.

Try the following modification of the previous example:

```
String[] text = {"love","is", "in", "the", "air"};

Arrays.sort(text);

int search = Arrays.binarySearch(text, "is");

System.out.println(search);
```

The outcome, since the array is sorted, will be:

```
2

Process finished with exit code 0
```

It is only a coincidence in this case that "`is`" happens to be in the same place in the unsorted and the sorted versions of the array. Making use of the tools you've been learning about, it should be possible for you to create an algorithm that can iterate throughout an array and count all the existing items, even if they are repeated, as well as locating their positions within the array. See *Activity 1*, *Searching for Multiple Occurrences in an Array* in this chapter, where we challenge you to write such a program.

You can also transform objects of the `java.util.Arrays` class into strings with the `Arrays.toString()` method, as we saw at the beginning of this section, into a list with `Arrays.asList()` (we will see this in a later section, as well as in `Example05`) or into a set with `Arrays.setAll()`.

Arrays and collections play important roles in software development. This section of the chapter dives into the differences between them as well as how they can be used together. If you search the internet for the relationship between these two constructs, most references you find will be focused on the differences, such as:

- Arrays have fixed sizes, while collections have variable sizes.
- Arrays can hold objects of any kind, but also primitives; collections cannot contain primitives.
- Arrays will hold homogeneous elements (elements that are all the same nature), while collections can hold heterogeneous elements.
- Arrays have no underlying data structure, while collections are implemented using standard structures.

If you know the amount of data you are going to be dealing with, arrays are the preferred tool, mainly because arrays perform better than lists or sets in such cases. However, there will be countless occasions when you don't know the amount of data you will be dealing with, which is where lists will be handy.

Also, arrays can be used to programmatically populate collections. We will be doing this throughout this chapter as a way of saving you the time of having to manually type all the data that will end up inside a collection, for example. The following example shows how to populate a set using an array:

```
import java.util.*;
public class Example05 {
    public static void main(String[] args) {
        Integer[] myArray = new Integer[] {3, 25, 2, 79, 2};
        Set<Integer> mySet = new HashSet<>(Arrays.asList(myArray));
        System.out.println(mySet);
    }
}
```

Note

While it is also possible to declare the set without explicitly declaring the type of data to be contained in it, explicit declaration, as we've done here, will help you avoid any warning messages from the compiler.

In this program, there is an array of `Integer` used to initialize an object of the `HashSet` class, which is later printed out.

The outcome of this example is:

```
[2, 3, 25, 79]

Process finished with exit code 0
```

The previous code listing shows a couple of interesting things. First of all, you will notice that the output to the program is sorted; that is because the conversion of the array to a list using `Arrays.asList()` will make the dataset inherit the properties of a list, which means that it will be sorted. Also, since the data has been added to a set and sets do not include duplicates, duplicate number two is left out.

It is important to note that with collections, you can specify the type to be stored. As such, there would be a difference between the declaration in the previous example, where we displayed a generic declaration, and what follows. The type is declared here using the name given within angle brackets, `<>`. In this case, it is `<Integer>`. You could rewrite the instantiation of the object as follows:

```
Set<Integer> mySet = new HashSet<Integer>(Arrays.asList(myArray));
```

You will see that the result of executing the program will be the same.

Activity 1: Searching for Multiple Occurrences in an Array

Write a program that will count multiple occurrences of a certain word in an array of strings, where each one of the objects is a single word. Use the following array, a famous quote by Frank Zappa, as a point of departure:

```
String[] text = {"So", "many", "books", "so", "little", "time"};
```

The word to search for is `so`. but you will have to consider that it shows up twice and that one instance is not in lowercase. As a hint, the method to compare two strings without looking at the specific casing of any of the letters in them
is `text1.compareToIgnoreCase(text2)`. To do so, perform the following steps:

1. Create the `text` array.
2. Create the variable that contains the word to be searched for: `so`
3. Initialize the variable `occurrence` to -1.
4. Create a for loop to iterate through the array to check for the occurrence.
5. Print the total count and the positions of the occurrences.

That will give the following result:

```
Found query at: 0

Found query at: 3

Found: 2 coinciding words

Process finished with exit code 0
```

Note

The solution for this activity can be found via this link.

Sets

Sets within the collections framework are the programmatic equivalent of mathematical sets. This means that they can store objects of a specific type while avoiding duplicates. In the same way, sets offer methods that will let you handle data as you would in mathematics. You can add objects to a set, check whether a set is empty, combine the

elements of two sets to add all their elements into a single set, see what objects coincide with each other between two sets, and calculate the difference between two sets.

In the `java.util.Sets` class, we find three interfaces used to represent sets: `HashSet`, `TreeSet`, and `LinkedHashSet`. The differences between them are straightforward:

- `HashSet` will store data without guaranteeing the order of iteration.
- `TreeSet` orders a set by value.
- `LinkedHashSet` orders a set by arrival time.

Each of these interfaces is meant to be used under specific circumstances. Let's look at a couple of examples of sets, departing from the one in `Example05`, and look at how we can add other methods to check how to operate sets. The first step is populating a set from an array. There are several methods for doing so; let's use the one that is probably the quickest to implement:

```
import java.util.*;
public class Example06 {
    public static void main(String[] args) {
        String[] myArray = new String[] {"3", "25", "2", "79", "2"};
        Set<String> mySet = new HashSet<>();
        Collections.addAll(mySet, myArray);
        System.out.println(mySet);
    }
}
```

The above line of code shows how to add all the elements of the array to the set; when printing the results, we get:

```
[2, 79, 3, 25]
```

```
Process finished with exit code 0
```

Please note that the order of the resulting print may vary for you. As explained earlier, `HashSet`, because of the way it is implemented, cannot guarantee any sorting of the content. If you performed the following example using `Integer` instead of `String` for the data, it would end up being sorted:

```
import java.util.*;

public class Example07 {
    public static void main(String[] args) {
        Integer[] myArray = new Integer[] {3, 25, 2, 79, 2};
        Set<Integer> mySet = new HashSet<>();
        Collections.addAll(mySet, myArray);
        System.out.println(mySet);
    }
}
```

The result of this program is:

```
[2, 3, 25, 79]

Process finished with exit code 0
```

This means that the results end up being sorted, even if we don't request it.

Note

The fact that the set in this example is sorted is a mere coincidence. Please be aware that this may not be the case in other situations. `Example08` will show the union operation between two sets, and there the data will not be sorted.

Working with sets involves working with packages of data and performing operations with them. The union operation for two sets is displayed in the following example:

```
import java.util.*;
public class Example08 {
    public static void main(String[] args) {
        Integer[] numbers1 = new Integer[] {3, 25, 2, 79, 2};
        Integer[] numbers2 = new Integer[] {7, 12, 14, 79};
        Set<Integer> set1 = new HashSet<>();
        Collections.addAll(set1, numbers1);
        Set<Integer> set2 = new HashSet<>();
        Collections.addAll(set2, numbers2);
        set1.addAll(set2);
        System.out.println(set1);
    }
}
```

This program will print, as its output, the resulting set from the union of the two sets described by the two arrays at the beginning of the main method of the example:

```
[2, 3, 7, 25, 12, 14, 79]
Process finished with exit code 0
```

Besides `HashSet`, we also find `TreeSet`, and here is where data will be sorted by value. Let's simply change the types of the sets in the previous example and see the result:

```
Set<Integer> set1 = new TreeSet<>();
Collections.addAll(set1, numbers1);
Set<Integer> set2 = new TreeSet<>();
Collections.addAll(set2, numbers2);
```

This, when changed in the previous example, will give the following sorted set as a result:

```
[2, 3, 7, 12, 14, 25, 79]
```

You might be wondering about the pros and cons of using each type of set. When sorting, you are trading speed for tidiness. Therefore, if you are working with large sets of data and speed is a concern, you will have to decide whether it is more convenient to have the system operate faster, or have the results sorted, which would allow faster binary searches through the dataset.

Given this last modification, we could perform other operations with the data, such as the intersection operation, which is invoked with the `set1.retainAll(set2)` method. Let's see it in action:

```
import java.util.*;
public class Example09 {
    public static void main(String[] args) {
        Integer[] numbers1 = new Integer[] {3, 25, 2, 79, 2};
        Integer[] numbers2 = new Integer[] {7, 12, 14, 79};
        Set<Integer> set1 = new TreeSet<>();
        Collections.addAll(set1, numbers1);
        Set<Integer> set2 = new TreeSet<>();
        Collections.addAll(set2, numbers2);
        set1.retainAll(set2);
        System.out.println(set1);
    }
}
```

For the output, given that the arrays are used to populate the arrays, we will get only those numbers that exist in both arrays; in this case, it is just the number `79`:

```
[79]

Process finished with exit code 0
```

The third type of set, `LinkedHashSet`, will sort the objects in order of their arrival. To demonstrate this behavior, let's make a program that will add elements to the set one by one using the `set.add(element)` command.

```
import java.util.*;
public class Example10 {
    public static void main(String[] args) {
        Set<Integer> set1 = new LinkedHashSet<>();
        set1.add(35);
        set1.add(19);
        set1.add(11);
        set1.add(83);
        set1.add(7);
        System.out.println(set1);
    }
}
```

When running this example, the result will be sorted by the way the data arrived in the set:

```
[35, 19, 11, 83, 7]

Process finished with exit code 0
```

For the sake of experimentation, use the next 2 minutes to chalk out the set construction into `HashSet` once more:

```
Set set1 = new HashSet();
```

The result of this modified program is uncertain. For example, we get:

```
[35, 19, 83, 7, 11]

Process finished with exit code 0
```

This is, again, an unsorted version of the same set of data.

To close our explanation of the possible methods that you can use with sets, let's use `LinkedHashSet` to run an experiment where we will find the difference between two sets.

```
import java.util.*;
public class Example11 {
    public static void main(String[] args) {
        Set<Integer> set1 = new LinkedHashSet<>();
        set1.add(35);
        set1.add(19);
        set1.add(11);
        set1.add(83);
        set1.add(7);
        Set<Integer> set2 = new LinkedHashSet<>();
        set2.add(3);
        set2.add(19);
        set2.add(11);
        set2.add(0);
        set2.add(7);
```

```
        set1.removeAll(set2);
        System.out.println(set1);
    }
}
```

In this case, both sets are slightly different, and by determining the difference, the algorithm behind `set1.removeAll(set2)` will look for the occurrences of each item in `set2` within `set1` and eliminate them. The result of this program is:

```
[35, 83]
Process finished with exit code 0
```

Finally, if you just want to check whether the whole of a set is contained within another set, you can call the `set1.containsAll(set2)` method. We'll leave that for you to explore – just be aware that the method simply responds with a Boolean stating whether the statement is true or false.

Lists

Lists are ordered collections of data. Unlike sets, lists can have repeated data. Having data contained within lists allows you to perform searches that will give the locations of certain objects within a given list. Given a position, it is possible to directly access an item in a list, add new items, remove items, and even add full lists. Lists are sequential, which makes them easy to navigate using iterators, a feature that will be explored in full in a later section in the chapter. There are also some methods for performing range-based operations on sublists.

There are two different list implementations: `ArrayList` and `LinkedList`. Each of them is ideal depending on the circumstances. Here, we will work with `ArrayList` mainly. Let's start by creating and populating an instance, then search for a certain value, and given its location within the list, we'll print out the value.

```
import java.util.*;
```

```
public class Example12 {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(35);
        list.add(19);
        list.add(11);
        list.add(83);
        list.add(7);
        System.out.println(list);
        int index = list.indexOf(19);
        System.out.println("Find 19 at: " + index);
        System.out.println("Component: " + list.get(index));
    }
}
```

The output of this example is:

```
[35, 19, 11, 83, 7]
Find 19 at: 1
Component: 19
Process finished with exit code 0
```

The `indexOf` method informs you about the location of an object passed to the method as a parameter. It's sibling method, `lastIndexOf`, reports the location of the last occurrence of an object in the list.

You should look at a list as a series of nodes connected by links. If one of the nodes is eliminated, the link that used to point to it will be redirected to the following item in the list. When adding nodes, they will be attached by default at the end of the list (if they are

not duplicated). As all the nodes in the collection are of the same type, it should be possible to exchange the locations of two nodes in a list.

Let's experiment with removing an item from a list and ascertaining the locations for objects located immediately before and after the removed item:

```
import java.util.*;
public class Example13 {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(35);
        list.add(19);
        list.add(11);
        list.add(83);
        list.add(7);
        System.out.println(list);
        int index = list.lastIndexOf(83);
        System.out.println("Before: find 83 at: " + index);
        list.remove(index - 1);
        System.out.println(list);
        index = list.lastIndexOf(83);
        System.out.println("After: find 83 at: " + index);
    }
}
```

This program creates a list, prints it out, looks for a node in the list, and prints its location. Then, it removes an item in the list and repeats the previous process to show that the node has been removed from the list. This is a clear difference from the case

with arrays, where it is not possible to remove items from them, and thus it is not possible to change their size. Observe the output of the previous example:

```
[35, 19, 11, 83, 7]
Before: find 83 at: 3
[35, 19, 83, 7]
After: find 83 at: 2
Process finished with exit code 0
```

It is also possible to change the content of a node. In the previous example, instead of removing a node, change `list.remove(index-1);` to the following and check the outcome:

```
list.set(index - 1, 99);
```

The final array will have substituted `11` for `99`.

If instead of deleting one node, you wanted to empty the whole list, the command to the issue would be:

```
list.clear();
```

Using `subList()`, an operator that generates lists from lists, it is possible to, for example, delete a range of cells within a list. See the following example, which deletes part of a string array, changing its meaning when printing it:

```
import java.util.*;
public class Example14 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("No");
        list.add("matter");
```

```
        list.add("what");
        list.add("you");
        list.add("do");
        System.out.println(list);
        list.subList(2,4).clear();
        System.out.println(list);
    }
}
```

Look at the following result:

```
[No, matter, what, you, do]
[No, matter, do]
Process finished with exit code 0
```

The `list` object has been modified by running the example code so that it becomes shorter. The two index numbers used in the `subList()` method are the places in the list where the method starts and stops. The result of `subList()` can also be assigned to a different variable of the same `List` type, resulting in a reduced copy of the list in the code, after performing the `subList()` operation.

Look at the following modification in the latest code listing:

```
List list1 = list.subList(2,4);
System.out.println(list1);
```

This will print out the list that was made of the nodes that were deleted in the previous example.

There are a lot of interesting algorithms within the collections framework that offers relevant functionality for operating with lists:

- `sort`: Put the elements of a list in a certain order.
- `shuffle`: Randomize the locations of all objects in a list.
- `reverse`: Invert the order of a list.
- `rotate`: Move objects to the end of a list, and when they reach the end, have them show up at the other end.
- `swap`: Swap two elements with one another.
- `replaceAll`: Replace all occurrences of an element in a list using a parameter.
- `fill`: Fill the content of a list with one value.
- `copy`: Make more instances of a list.
- `binarySearch`: Perform optimized searches within a list.
- `indexOfSubList`: Search for the occurrence of a piece (a set of consecutive nodes) of a list.
- `lastIndexOfSubList`: Search for the last occurrence of a piece of a list.

> Note
>
> Lists generated from arrays using `Arrays.asList()` do not behave in the same way as the objects of the `List` class described in this section. The lists coming from arrays have a fixed length, which means that elements cannot be removed from the array. The reason for this is that `java.util.Arrays` implement its own `ArrayList` class inside the package, one that is different from the one in the collections framework. Confusing, isn't it?

Exercise 1: Creating the AnalyzeInput Application

In this exercise, we will create a new application that will respond to the CLI by storing whatever strings are provided to it, then run some statistical operations on the data, such as word counting (determining the most frequent word or the most frequent letter, and so on). The intent is to give you an idea of how to use the collections framework instead of other tools to do such operations. This time, we will do something special; instead of getting the data from the CLI as arguments to the script, we will use the `java.io.Console` API, which allows the reading of different types of strings from the terminal, such as usernames (plain strings) and passwords. The goal of this application is to read the input until a line with only the "`*`" symbol (asterisk) is captured. Once the

termination symbol is entered, the text will be processed, and the statistics will be delivered to the terminal:

1. Open IntelliJ and create a new Java program using the CLI template. Name the project `AnalyzeInput`.
2. Start by creating a simple program that can read a line from the terminal and printing it out:

```
3.   import java.io.Console;
4.   public class AnalyzeInput {
5.     public static void main(String[] args) {
6.       Console cons;
7.       String line = "";
8.         if ((cons = System.console()) != null && (line =
   cons.readLine()) !=         null) {
9.         System.out.println("You typed: " + line);
10.      }
11.    }
   }
```

12. Execute the program from the CLI by calling `java AnalyzeInput` from the right folder and interact with it:

```
13.  usr@localhost:~/IdeaProjects/ch04/out/production/ch04$ java
   AnalyzeInput
14.  hej this is an example
   You typed: hej this is an example
```

15. You must import `java.io.Console`, which allows you to instantiate objects of the `Console` class. You can also see the call to `cons = System.console()`, which will make sure that the terminal is ready for you to read the data, and `line = cons.readLine()`, which will ensure that when hitting the *Enter* key on the keyboard, the resulting data is not empty.

16. The next step is storing the data we are capturing in a collection. Since we don't know the size this could be, we should be using `ArrayList <String>` to store the data. Also, to store data for as long as we want, we can modify the `if` statement and make it into a `while` loop. Finally, use the `add` method to add the lines into a list (note that the following code listing will never exit, so bear with us and do not execute it yet):

```
17.  import java.util.*;
18.  import java.io.Console;
19.  public class Exercise01 {
20.    public static void main(String[] args) {
21.      ArrayList <String> text = new ArrayList<>();
22.      Console cons;
23.      String line = "";
24.      while ((cons = System.console()) != null && (line =
    cons.readLine())       != null) {
25.      text.add(line);
26.      }
27.      System.out.println("You typed: " + text);
28.    }
   }
```

29. Modify the `while` loop to include the condition we established for finishing the data capture process – the arrival of a line with only an asterisk symbol:

```
30.          while (!line.equals("*")
31.              && (cons = System.console()) != null
             && (line = cons.readLine()) != null) {
```

32. The outcome will happen only when you type the asterisk symbol alone in a line, as seen in this log while interacting with the program:

```
usr@localhost:~/IdeaProjects/ch04/out/production/ch04$ java
AnalyzeInput
this is the array example
until you type *
alone in a line
*
You typed: [this is the array example, until you type *, alone in a
line, *]
```

38. Since we used `ArrayList` to store the different strings, you could be typing until you exhaust the computer's memory. Now it is possible to execute some commands to work with the strings. The first step will be turning the whole of the text into a list. This will require going through the different strings and splitting them into parts that will be added to a larger list. The easiest trick is to use the `split()` method using a whitespace character as a separator. Modify the `main` method to look like the following, and you will see that the result is now a list with all the words separated as single nodes in the list:

```
    public static void main(String[] args) {
        ArrayList <String> text = new ArrayList<>();
        Console cons;
        String line = "";
        while (!line.equals("*")
            && (cons = System.console()) != null
            && (line = cons.readLine()) != null) {
                List<String> lineList = new
                  ArrayList<>(Arrays.asList(line.split(" ")));
            text.addAll(lineList);
        }
```

```
50.             System.out.println("You typed: " + text);
    }
```

51. Having all the data stored in this way allows for the use of a lot of the methods available in the collections framework that will let you do operations with data. Let's start by counting all the words in the text (including the closing symbol, "`*`"). Just add the following at the end of the `main` method:

```
System.out.println("Word count: " + text.size());
```

The result of this exercise is a program that is ready to be used for further analysis of the data. But in order to continue doing so, we need to make use of a tool that has not yet been introduced—the iterator. We will come back to this example later in the chapter and finish off the application by adding some extra functionality to it.

Maps

The collections framework offers one more interface, `java.util.Map`, which can be used when dealing with data that is stored as key-value pairs. This type of data storage is becoming more and more relevant as data formats such as JSON are slowly taking over the internet. JSON is a data format that is based on having data stored in the form of nested arrays that always respond to the key-value structure.

Having data organized in this way offers the possibility of having a very simple way to look for data – by means of the keys instead of using, for example, an index, as we would do in an array. Keys are the way we can identify the block of data we are looking for within a map. Let's look at a simple example of a map before looking at alternatives to maps:

The following example shows how to create a simple map and how to print some messages based on the information available within it. The first thing that you will notice in comparison to other interfaces in the collections framework is that we do not *add* elements to the map – we *put* elements in the map. Also, elements have two parts: the **key** (in our case, we are using strings) and the **value** (which can be heterogeneous in nature):

```
import java.util.*;
public class Example15 {
    public static void main(String[] args) {
        Map map = new HashMap();
        map.put("number", new Integer(1));
        map.put("text", new String("hola"));
        map.put("decimal", new Double(5.7));
        System.out.println(map.get("text"));
        if (!map.containsKey("byte")) {
            System.out.println("There are no bytes here!");
        }
    }
}
```

This program will give the following result:

```
hola
There are no bytes here!
Process finished with exit code 0
```

Since there is no key named "`bytes`" in the code, the `maps.containsKey()` method will answer accordingly, and the program will inform the user about it. The main methods available in this interface are:

- `put` (Object key, Object value)
- `putAll` (Map map)
- `remove` (Object key)
- `get` (Object key)
- `containsKey` (Object key)

- `keySet()`
- `entrySet()`

All but the last two are self-explanatory. Let's focus on augmenting our previous example to see what those two methods do. Make the following addition to the code to see what `keySet()` and `entrySet()` have to offer:

```
System.out.println(map.entrySet());

System.out.println(map.keySet());
```

The outcome of the modified code listing will be:

```
hola

There are no bytes here!

[number=1, text=hola, decimal=5.7]

[number, text, decimal]

Process finished with exit code 0
```

In other words, `entrySet()` will print the whole map using the key = value formula, while `keySet()` will respond with the set of keys within the map.

Note

You might have realized this by now: keys must be unique – there cannot be two of the same keys in a map.

We will not go deeper into maps at this point because they are, to an extent, a repetition of what we saw with sets. There are three different classes for maps: `HashMap`, `TreeMap`, and `LinkedHashMap`. The last two are put in order, while the first one is neither sorted nor arranged in order of arrival. You should use these classes according to your needs.

Iterating through Collections

Earlier in this chapter, when working with *Exercise 01, Creating the AnalyzeInput Application* we stopped when we were about to make searches through the data. We made it to the point where we had to iterate through the data and look for characteristics such as word frequency.

Iterators are used in Java to browse through collections. Let's look at a simple example that involves extracting the elements from a simple list one by one and printing them out.

```
import java.util.*;
public class Example16 {
    public static void main(String[] args) {
        List<Integer> array = new ArrayList<>();
        array.add(5);
        array.add(2);
        array.add(37);
        Iterator<Integer> iterator = array.iterator();
        while (iterator.hasNext()) {
                // point to next element
                int i = iterator.next();
                // print elements
                System.out.print(i + " ");
        }
    }
}
```

The output of this program is:

```
5 2 37
```

```
Process finished with exit code 0
```

Iterators such as this one are the most generic ones in the collections framework and can be used with lists, sets, queues, and even maps. There are other less-broad implementations of the iterators that allow for different ways to browse through data, for example, in lists. As you saw in the latest code listing, the `iterator.hasNext()` method checks whether there is a node after the one we are at in the list. When starting the iterator, the object points to the first element in the list. Then, `hasNext()` responds with a Boolean stating whether there are more nodes hanging from it.
The `iterator.next()` method will move the iterator to the following node in the collection. This kind of iterator does not have the possibility of going back in the collection; it can only move forward. There is one final method in the iterator, called `remove()`, which will eliminate the current element that the iterator is pointing to from the collection.

If we used `listIterator()` instead, we would have had a lot more options for navigating collections, such as adding new elements and changing elements. The following code listing demonstrates how to go through a list, add elements, and modify them. `listIterator` works only with lists:

```
import java.util.*;
public class Example17 {
    public static void main(String[] args) {
        List <Double> array = new ArrayList<>();
        array.add(5.0);
        array.add(2.2);
        array.add(37.5);
        array.add(3.1);
        array.add(1.3);
        System.out.println("Original list: " + array);
        ListIterator listIterator = array.listIterator();
```

```
        while (listIterator.hasNext()) {
            //  point to next element
            double d = (Double) listIterator.next();
            // round up the decimal number
            listIterator.set(Math.round(d));
        }
        System.out.println("Modified list: " + array);
    }
}
```

In this example, we create a list of `Double`, iterate through the list, and round up each of the numbers. The outcome of this program is:

```
Original list: [5.0, 2.2, 37.5, 3.1, 1.3]
Modified list: [5, 2, 38, 3, 1]
Process finished with exit code 0
```

By calling `listIterator.set()`, we modify each of the items in the list and the second `System.out.println()` command shows where the numbers have been rounded up or down.

The final iterator example we are going to see in this section is a trick to iterate through a map. This could come in handy in scenarios where you want to perform some operations on data within a map. By using the `entrySet()` method – which returns a list – it is possible to have an iterator over a map. More specifically, `entrySet()` returns an object that can have an iterator assigned (since it is a list). This iterator is an instance of the `Map.Entry` class, in which objects are composed of key-value pairs. See the following example to understand how this works:

```
import java.util.*;
public class AnalyzeInput {
```

```
    public static void main(String[] args) {
        Map map = new HashMap ();
        map.put("name", "Kristian");
        map.put("family name", "Larssen");
        map.put("address", "Jumping Rd");
        map.put("mobile", "555-12345");
        map.put("pet", "cat");
        Iterator <Map.Entry> iterator = map.entrySet().iterator();
        while (iterator.hasNext()) {
            Map.Entry entry = iterator.next();
            System.out.print("Key = " + entry.getKey());
            System.out.println( ", Value = " + entry.getValue());
        }
    }
}
```

This program will iterate through a map and print the contents as they were stored in `HashMap`. Remember that these types of objects are not sorted in any specific way. You can expect an output like the following:

```
Key = address, Value = Jumping Rd
Key = family name, Value = Larssen
Key = name, Value = Kristian
Key = mobile, Value = 555-12345
Key = pet, Value = cat
Process finished with exit code 0
```

Given that we now have ways to iterate through collections, we can move on to an exercise that picks up where we left off: iterating through a list for data analysis.

Exercise 2: Bringing Analytics into the AnalyzeInput Application

We are going to start from where we left off at the end of *Exercise 1*, *Creating the AnalyzeInput Application*. We managed to capture the text typed in the terminal and store it as a list of strings. This time, we are going to use a method from the collections framework called `frequency`, which will respond with the number of times a certain object can be found inside a list. As words could be repeated in a sentence, we first need to figure out a way to extract the unique elements in a list:

1. Sets are objects in the collections framework that keep only one copy of each element. We saw an example of this earlier in the chapter. We will create a `HashSet` instance and copy all the elements from the list into it. This will automatically eliminate duplicates:

```
2.    Set <String> textSet = new HashSet<>();
      textSet.addAll(text);
```

3. The next step, now that we have the set, is to create an iterator that will check how many copies of each element from the set can be found in the list:

```
Iterator iterator = textSet.iterator();
```

4. Using the same technique that we saw in previous examples for how to iterate through a set, we will find the next node in the set and check in the list for the frequency of the string stored in the node:

```
5.    while (iterator.hasNext()) {
6.        //point to next element
7.        String s = (String) iterator.next();
8.        // get the amount of times this word shows up in the text
9.        int freq = Collections.frequency(text, s);
10.       // print out the result
```

```
11.       System.out.println(s + " appears " + freq + " times");
    }
```

Note

The final code can be referred at: https://packt.live/2BrplvS.

12. The outcome will depend on the kind of text you type. For the sake of testing, try the following (we will stick to this data entry for the rest of the chapter – you can copy and paste it to the terminal each time you call the application):

```
13.  this is a test
14.  is a test
15.  test is this
    *
```

16. The full outcome of this input will be:

```
17.  You typed: [this, is, a, test, is, a, test, test, is, this, *]
18.  Word count: 11
19.  a appears 2 times
20.  test appears 3 times
21.  this appears 2 times
22.  is appears 3 times
    * appears 1 times
```

While the result is correct, it is not easy to read through. Ideally, results should be sorted. For example, by descending values of frequency, so that it is easy to see at a glance the most and least frequent words. This is the time to make yet another stop in the exercise as we need to introduce the idea of sorting before we move on with it.

Sorting Collections

As we have seen, there are some classes in the collections framework that force the items within them to be sorted. Examples of that are `TreeSet` and `TreeMap`. The aspect to explore in this section is how to use existing sorting mechanisms for lists, but also for cases that have datasets with more than one value per data point.

The exercise we are doing throughout this chapter is a good example of a case where there are data points with more than one value. For each data point, we need to store the word for which we are calculating the frequency and the frequency itself. You might think that a good technique to sort that out is by storing the information in the form of maps. The unique words could be the keys, while the frequencies could be the values. This could be achieved by modifying the final part of the previous program to look like this:

```
Map map = new HashMap();
while (iterator.hasNext()) {
    //  point to next element
    String s = (String) iterator.next();
    // get the amount of times this word shows up in the text
    int freq = Collections.frequency(text, s);
    // print out the result
    System.out.println(s + " appears " + freq + " times");
    // add items to the map
    map.put(s, freq);
}
TreeMap mapTree = new TreeMap();
mapTree.putAll(map);
System.out.println(mapTree);
```

While this is an interesting and simple approach to sorting (copying the data into a structure that is sorted by nature), it presents the problem that data is sorted by key and not by value, as the following result of the previous code highlights:

```
Word count: 11
a appears 2 times
test appears 3 times
this appears 2 times
is appears 3 times
* appears 1 times
{*=1, a=2, is=3, test=3, this=2}
```

So, if we want to sort these results by value, we need to figure out a different strategy.

But let's step back for a second and analyze what tools are offered in the collections framework for sorting. There is a method called `sort()` that can be used to sort lists. An example of this is as follows:

```
import java.util.*;
public class Example19 {
    public static void main(String[] args) {
        List <Double> array = new ArrayList<>();
        array.add(5.0);
        array.add(2.2);
        array.add(37.5);
        array.add(3.1);
        array.add(1.3);
        System.out.println("Original list: " + array);
        Collections.sort(array);
```

```
        System.out.println("Modified list: " + array);
    }
}
```

The result of this program is:

```
Original list: [5.0, 2.2, 37.5, 3.1, 1.3]
Modified list: [1.3, 2.2, 3.1, 5.0, 37.5]
Process finished with exit code 0
```

Given a list, we could sort it this way just fine; it would even be possible to navigate through it backward using `listIterator` to sort a list in descending order. However, these methods do not solve the issue of sorting data points with multiple values. In such a case, we would need to create a class to store our own key-value pair. Let's see how to implement this by continuing with the exercise we have been dealing with throughout the chapter.

Exercise 3: Sort the Results from the AnalyzeInput Application

We now have a program that, given some input text, identifies some basic characteristics of the text, such as the number of words in the text or the frequency of each of the words. Our goal is to be able to sort the results in descending order to make them easier to read. The solution will require the implementation of a class that will store our key-value pairs and make a list of objects from that class:

1. Create a class containing the two data points: the word and its frequency. Implement a constructor that will take values and pass them to class variables. This will simplify the code later:

```
2.   class DataPoint {
3.       String key = "";
4.       Integer value = 0;
```

```
5.        // constructor
6.        DataPoint(String s, Integer i) {
7.            key = s;
8.            value = i;
9.        }
    }
```

10. When calculating the frequency for each word, store the results in a newly created list of objects of the new class:

```
11.           List <DataPoint> frequencies = new ArrayList <> ();
12.           while (iterator.hasNext()) {
13.           //point to next element
14.           String s = (String) iterator.next();
15.           // get the amount of times this word shows up in the text
16.           int freq = Collections.frequency(text, s);
17.           // print out the result
18.           System.out.println(s + " appears " + freq + " times");
19.           // create the object to be stored
20.           DataPoint datapoint = new DataPoint (s, freq);
21.           // add datapoints to the list
22.           frequencies.add(datapoint);
    }
```

23. Sorting is going to require the creation of a new class using the `Comparator` interface, which we are just introducing now. This interface should implement a method that will be used to run comparisons within the objects in the array. This new class must implement `Comparator <DataPoint>` and include a single method called `compare()`. It should have two objects of the class being sorted as parameters:

```
class SortByValue implements Comparator<DataPoint>
{
    // Used for sorting in ascending order
    public int compare(DataPoint a, DataPoint b)
    {
        return a.value - b.value;
    }
}
```

31. The way we call the `Collections.sort()` algorithm using this new comparator is by adding an object of that class as a parameter to the `sort` method. We instantiate it directly in the call:

```
Collections.sort(frequencies,new SortByValue());
```

32. This will sort the frequencies list in ascending order. To print the results, it is no longer valid to make a direct call to `System.out.println(frequencies)` because it is now an array of objects and it will not print the contents of the data points to the terminal. Iterate through the list in the following way instead:

```
System.out.println("Results sorted");
for (int i = 0; i < frequencies.size(); i++)
    System.out.println(frequencies.get(i).value
                       + " times for word "
                     + frequencies.get(i).key);
```

37. If you run the program using the same input that we have been using for the last couple of examples, the outcome will be:

```
Results sorted
1 times for word *
2 times for word a
2 times for word this
```

```
42.  3 times for word test
     3 times for word is
```

43. Our goal is to sort the results in descending order and, to do that, we will need to add one more thing to the call to the `sort` algorithm. When instantiating the `SortByValue()` class, we need to tell the compiler that we want the list to be sorted in reverse order. The collections framework already has a method for this:

```
Collections.sort(frequencies, Collections.reverseOrder(new
SortByValue()));
```

Note

For the sake of clarity, the final code can be referred at: https://packt.live/2W5qhzP.

44. A full interaction path with this program, from the moment we call it to include the data entry, would be as follows:

```
45.  user@localhost:~/IdeaProjects/ch04/out/production/ch04$ java
     AnalyzeInput
46.  this is a test
47.  is a test
48.  test is this
49.  *
50.  You typed: [this, is, a, test, is, a, test, test, is, this, *]
51.  Word count: 11
52.  a appears 2 times
53.  test appears 3 times
54.  this appears 2 times
55.  is appears 3 times
56.  * appears 1 times
```

```
57.  Results sorted
58.  3 times for word test
59.  3 times for word is
60.  2 times for word a
61.  2 times for word this
    1 times for word *
```

Properties

Properties in the collections framework are used to maintain lists of key-value pairs where both are of the `String` class. Properties are relevant when obtaining environmental values from the operating system, for example, and are the grounding class for many other classes. One of the main characteristics of the `Properties` class is that it allows the definition of a default response in the case of a search for a certain key not being satisfactory. The following example highlights the basics of this case:

Example20.java

```
1  import java.util.*;
2
3  public class Example20 {
4
5     public static void main(String[] args) {
6         Properties properties = new Properties();
7         Set setOfKeys;
8         String key;
9
10        properties.put("OS", "Ubuntu Linux");
```

```
11          properties.put("version", "18.04");
12          properties.put("language", "English (UK)");
13
14          // iterate through the map
15          setOfKeys = properties.keySet();
```

https://packt.live/2N0CzoS

Before diving into the results, you will notice that in properties, we put rather than add new elements/nodes. This is the same as we saw with maps. Also, you will have noticed that to iterate, we used the `keySet()` technique that we saw when iterating through maps earlier. Finally, the particularity of `Properties` is that you can set a default response in the case of the searched-for property not being found. This is what happens in the example when searching for **keyboard layout**—it was never defined, so the `getProperty()` method will answer with its default message without crashing the program.

The result of this program is:

```
version = 18.04
OS = Ubuntu Linux
language = English (UK)
keyboard layout = not found
Process finished with exit code 0
```

Another interesting method to be found in the `Properties` class is the `list()`; it comes with two different implementations that allow you to send the contents of a list to different data handlers. We can stream the whole properties list to a `PrintStreamer` object, such as `System.out`. This offers a simple way of displaying what is in a list without having to iterate through it. An example of this follows:

```
import java.util.*;
```

```
public class Example21 {
    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put("OS", "Ubuntu Linux");
        properties.put("version", "18.04");
        properties.put("language", "English (UK)");
        properties.list(System.out);
    }
}
```

That will result in:

```
version=18.04
OS=Ubuntu Linux
language=English (UK)
Process finished with exit code 0
```

The `propertyNames()` method returns an `Enumeration` list, and by iterating through it, we will obtain the keys to the whole list. This is an alternative to creating a set and running the `keySet()` method.

```
import java.util.*;
public class Example22 {
    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put("OS", "Ubuntu Linux");
        properties.put("version", "18.04");
        properties.put("language", "English (UK)");
```

```
        Enumeration enumeration = properties.propertyNames();
        while (enumeration.hasMoreElements()) {
            System.out.println(enumeration.nextElement());
        }
    }
}
```

That will result in:

```
version
OS
language
Process finished with exit code 0
```

The final method we will introduce you to from `Properties` at this point is `setProperty()`. It will modify the value of an existing key, or will eventually create a new key-value pair if the key is not found. The method will answer with the old value if the key exists, and answer with `null` otherwise. The next example shows how it works:

```
import java.util.*;
public class Example23 {
    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put("OS", "Ubuntu Linux");
        properties.put("version", "18.04");
        properties.put("language", "English (UK)");
        String oldValue = (String) properties.setProperty("language",
"German");
```

```
        if (oldValue != null) {
            System.out.println("modified the language property");
        }
        properties.list(System.out);
    }
}
```

Here is the outcome:

```
modified the language property
-- listing properties --
version=18.04
OS=Ubuntu Linux
language=German
Process finished with exit code 0
```

Note

There are more methods in the `Properties` class that deals with storing and retrieving lists of properties to/from files. While this is a very powerful feature from the Java APIs, as we haven't yet introduced the use of files in this book, we will not discuss those methods here. For more information at this point, please refer to Java's official documentation.

Activity 2: Iterating through Large Lists

In contemporary computing, we deal with large sets of data. The purpose of this activity is to create a random-sized list of random numbers to perform some basic operations on data, such as obtaining the average.

1. To start, you should create a random list of numbers.

2. To compute the average, you could create an iterator that will go through the list of values and add the weighted value corresponding to each element.
3. The value coming from the `iterator.next()` method must be cast into a `Double` before it can be weighed against the total number of elements.

If you've implemented everything properly, the results of the averaging should similar to:

```
Total amount of numbers: 3246

Average: 49.785278826074396
```

Or, it could be:

```
Total amount of numbers: 6475

Average: 50.3373892275651
```

Note

The solution for this activity can be found via this link.

If you managed to make this program work, you should think about how to take advantage of being able to simulate large sets of data like this one. This data could represent the amount of time between different arrivals of data in your application, temperature data from the nodes in an Internet of Things network being captured every second. The possibilities are endless. By using lists, you can make the size of the dataset as endless as their working possibilities.

Summary

This chapter introduced you to the Java collections framework, which is a very powerful tool within the Java language that can be used to store, sort, and filter data. The framework is massive and offers tools in the form of interfaces, classes, and methods, some of which are beyond the scope of this chapter. We have focused on `Arrays`, `Lists`, `Sets`, `Maps`, and `Properties`. But there are others, such as queues and dequeues, that are worth exploring on your own.

Sets, like their mathematical equivalents, store unique copies of items. Lists are like arrays that can be extended endlessly and support duplicates. Maps are used when dealing with key-value pairs, something very common in contemporary computing, and do not support the use of two of the same keys. Properties work very much like `HashMap` (a specific type of `Map`) but offer some extra features, such as the listing of all their contents to streams, which simplifies the printing out of the contents of a list.

Some of the classes offered in the framework are sorted by design, such as `TreeHash` and `TreeMap`, while others are not. Depending on how you want to handle data, you will have to decide which is the best collection.

There are standard techniques for looking through data with iterators. These iterators, upon creation, will point to the first element in a list. Iterators offer some basic methods, such as `hasNext()` and `next()`, that state whether there is more data in the list and extract data from the list, respectively. While **hasNext()** and **next()** are common to all iterators, there are others, such as `listIterator`, that are much more powerful and allow, for example, the addition of new elements to a list while browsing through it.

We have looked at a chapter-long example that used many of these techniques, and we have introduced the use of the console to read data through the terminal. In the next chapter, we will cover exceptions and how to handle them.

Lambdas

4.1 Define A Lambda Function

Lambda expressions were added in *Java 8*. A lambda expression is a short block of code which takes in parameters and returns a value. Lambda is also known as *anonymous method*. It's a shorter way to write anonymous classes with only one method.

Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

he simplest lambda expression contains a single parameter and an expression:

```
parameter -> expression
```

To use more than one parameter, wrap them in parentheses:

```
(parameter1, parameter2) -> expression
```

Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a return statement.

```
(parameter1, parameter2) -> { code block }
```

Let's see a complete code example.

```
class Foo {
    Run | Debug
    public static void main(String[] args) {
        Message msg = () -> {
            System.out.println(x: "Hello lambda");
        };

        msg.message();

        Alert alert = (str) -> {
            String s = "Your are in " + str;
            return s;
        };

        String str = alert.alert(str: "Danger !");
        System.out.println(str);
    }
}

@FunctionalInterface
interface Message {
    public void message();
}

@FunctionalInterface
interface Alert {
    public String alert(String str);
}
```

4.2 Function as Value

An interesting part of lambda is it can be assigned to a variable. Let's have a look at the following example.


```
Message msg = () -> {
    System.out.println(x: "Hello lambda");
};
```

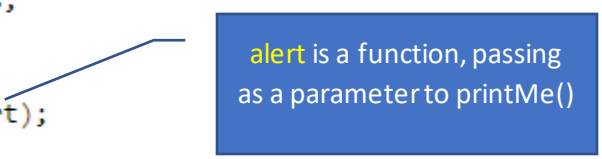
4.3 Passing Function as Function Parameter

What if we want to have the following method:

```
public void greet (action) {
  action();
}
```

This is passing a function parameter to a function. In Java this is now possible. Let's see a full example.

```
class Sample2 {
    Run | Debug
    public static void main(String[] args) {
        Alert alert = (str) -> {
            String s = "Your are in " + str;
            return s;
        };

        printMe(alert);
    }

    static void printMe(Alert a) {
        System.out.println(a.alert(str: "Safe place"));
    }
}
```



4.4 Lambda in Loop

The following sample shows how lambda can be used to print from **ArrayList** in a foreach loop.

```
class Sample3 {
    Run | Debug
    public static void main(String args[]){
      List<String> list=new ArrayList<String>();
          list.add(e: "Abu");
          list.add(e: "Ali");
          list.add(e: "Ahmad");
          list.add(e: "Vishal");
          list.add(e: "John");
          list.add(e: "Harry");

          list.forEach(
              (name)->System.out.println(name)
          );
    }
  }
```

Java Stream

- 5.1 Introduction
- 5.2 Creating Stream
- 5.3 Using Filter()
- 5.4 Using Map()
- 5.5 Using Sorted()
- 5.6 Using Reduce()

5.1 Introduction

A stream can be defined as *sequence of data*. Java 8 introduced the new Stream API, which bring functional programming to Java. With streams, Java programmers can now use a more *declarative style of writing programs* that you have previously only seen in functional programming languages or functional programming libraries.

Once you understand how to use Java Stream API, you will be able to work with **arrays** and **collections** by first learning how to create and close those streams.

Using streams, you can now *write more expressive programs* with fewer lines of code, and easily chain multiple operations on large lists. Streams also make it simple to parallelize your operations on lists — that is, should you have very large lists or complex operations.

Stream Source

Stream can be created from Collection, List, Sets, ints, longs, double, arrays, lines of a file.

Stream operations are either intermediate or terminal.

- **Intermediate operations** such as **filter**, **map** or **sort** return a stream so we can chain multiple intermediate operations.

- **Terminal operations** such as **forEach**, **collect** or **reduce** are either void or return a non-stream result.

Intermediate Operations

- Zero or more intermediate operations are allowed
- Order matters for large datasets: filter first, the sort or map
- For large datasets user **ParallelStream** to enable multiple threads
- Intermediate operations include:
    - anyMatch()
    - distinct()
    - filter()
    - findFirst()
    - flatMap()
    - map()
    - skip()
    - sorted()

Terminal Operations

- One terminal operation is allowed
- Collect saves the elements into collection
- Other options reduce the stream to a single summary element
- Operations include:
    - Count()
    - Max()
    - Min()
    - Reduce()
    - summaryStatictics()

In summary, java stream offer the following:

- Make you more efficient Java programmer
- Make heavy use of lambda expressions
- **ParallelStreams** make it very easy to multi-thread operations

5.2 Creating Stream

There are multiple ways of creating streams in Java; the simplest of these is by using the **Stream.of()** function. This function can take either a single object or multiple objects in varargs:

```
Stream<Object> objectStream = Stream.of(new Object());
```

If you have multiple objects in your stream, then use the varargs version:

```
Stream<Object> objectStream = Stream.of(new Object(), new Object(), new Object());
```

To create a stream from an array of items, you can use the `Arrays` class, just like the primitive versions of streams do

```
String[] stringArray = new String[]{"string1", "string2", "string3"};
Stream<String> stringStream = Arrays.stream(stringArray);
```

5.3 Using Filter()

filter: As the name suggests, this intermediate operation will return a subset of elements from the stream. It uses a predicate when applying the matching pattern, which is a functional interface that returns a Boolean. The easiest and most common way to implement this is using a lambda function:

```
Stream.of(...values: 1, 2, 3, 4, 5, 6)
.filter((i) -> { return i > 3; })
.forEach(System.out::println);
```

The result:

```
4
5
6
```

5.4 Using Map()

The map operation will apply a special function to every element of the stream and return the modified elements:

```
// map
Stream.of(...values: "5", "3", "8", "2").map((s) -> { return Integer.parseInt(s); })
.forEach((i) -> { System.out.println(i > 3); });
```

The result:

True

False

True

False

5.5 Using Sorted()

The sorted intermediate operation exists in two versions. The first version, without arguments, assumes that the elements of the map can be sorted in the natural order — implementing the Comparable interface. If they can't be sorted, then an exception will be thrown:

```
// sorted, natural order
Stream.of(...values: 1, 3, 6, 4, 5, 2)
.sorted()
.forEach((i) -> { System.out.print(i); });
```

The result:

123456

The second version of the sorted operation takes a Comparator as an argument, and will return the sorted elements accordingly:

5.6 Using Reduce()

In Java 8, the **Stream.reduce()** combine elements of a stream and produces a single value.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// 1st argument, init value = 0
int sum = Arrays.stream(numbers).reduce(identity: 0, (a, b) -> a + b);

System.out.println("sum : " + sum); // 55
```

a = running total
b = new arg
0 = initial val.

Result:

55

Annotation

- 7.1 Annotation as Metadata
- 7.2 Annotation on Method
- 7.3 Annotation on Class
- 7.4 Annotation on Variable
- 7.5 Annotation on Function Parameter

7.1 Annotation as Metadata

Java annotations are used to provide meta data for your Java code. Being meta data, Java annotations do not directly affect the execution of your code, although some types of annotations can actually be used for that purpose.

Java annotations were added to Java from Java 5. This text covers Java annotations as they look in Java 8, Java 9 and beyond. As far, Java annotations have not changed in later Java version, so this text should be valid for Java 8, 9, 10 and 11 programmers too.

Java annotations are typically used for the following purposes:

- Compiler instructions
- Build-time instructions
- Runtime instructions

Java has 3 built-in annotations that you can use to give instructions to the Java compiler. Java annotations can be be used at build-time, when you build your software project. The build process includes generating source code, compiling the source, generating XML files (e.g., deployment descriptors), packaging the compiled code and files into a JAR file etc. Building the software is typically done by an automatic build tool like Apache Ant or Apache Maven. Build tools may scan your Java code for specific annotations and generate source code or other files based on these annotations.

Normally, Java annotations are not present in your Java code after compilation. It is possible, however, to define your own annotations that are available at runtime. These annotations can then be accessed via Java Reflection, and used to give instructions to your program, or some third party API.

Built-in Java Annotations

Java comes with three built-in annotations which are used to give the Java compiler instructions. These annotations are:

- @Deprecated
- @Override
- @SuppressWarnings
- @Contended

Custom Java Annotation Example

Here is custom Java annotation example:

```
@interface MyAnnotation {

    String   value();

    String   name();
    int      age();
    String[] newNames();

}
```

This example defines an annotation called MyAnnotation which has four elements. Notice the @interface keyword. This signals to the Java compiler that this is a Java annotation definition.

Notice that each element is defined similarly to a method definition in an interface. It has a data type and a name. You can use all primitive data types as element data types. You can also use arrays as data type. You cannot use complex objects as data type.

To use the above annotation, you could use code like this:

```
@MyAnnotation(
    value="123",
    name="Azman",
    age=37,
    newNames={"Azman", "Zakaria"}
)
class MyClass {


}
```

7.2 Annotation on Method

The following example shows how to apply annotation on method.

```
@Getter
public String getVehicleName() {
    return this.vehicleName;
}
```

7.3 Annotation on Class

You can place Java annotations above classes, interfaces, methods, method parameters, fields and local variables. Here is an example annotation added above a class definition:

@Entity

public class Vehicle {

}

The annotation starts with the @ character, followed by the name of the annotation. In this case, the annotation name is Entity. The Entity annotation is an annotation I have made up. It doesn't have any meaning in Java.

7.4 Annotation on Variable

The following example shows how to apply annotation on a variable.

```
@Optional
List localNames = names;
```

7.5 Annotation on Function Parameter

The following example shows how to apply annotation on method parameter.

```
public void setVehicleName(@Optional vehicleName) {
    this.vehicleName = vehicleName;
}
```

Generic

- 8.1 Strong Type-Checking
- 8.2 Elimination of Cast
- 8.3 Code Reuse, Type Safety and Performance
- 8.4 Collection and Generic

8.1 Strong Type-Checking

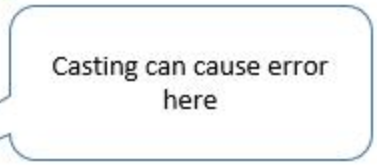
It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time. Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

The following code demonstrates problem-solving of this issue.

```
public class List {
    private int[] items = new int[10];
    private int count = 0;

    public void add(int item) {
        items[count++] = item;
    }

    public int get(int index) {
        return items[index];
    }

    Run | Debug
    public static void main(String[] args) {
        var list = new List();
        list.add(item: 100);

        var list2 = new List2();
        list2.add(item: "item 1");
        list2.add(item: 100);
        //Integer val = (int)list2.get(1); // this ok
        int val = (int)list2.get(index: 0); // this error
        System.out.println(val);
    }
}

class List2 {
    private Object[] items = new Object[10];
    private int count = 0;

    public void add(Object item) {
        items[count++] = item;
    }

    public Object get(int index) {
        return items[index];
    }
}
```



8.2 Elimination Of Cast

The following code demonstrates how the usage of generic type can eliminate of type casting.

```
public class GenericList<T> {
    private T[] items = (T[]) new Object[10];
    private int count;

    public void add(T item) {
        items[count++] = item;
    }

    public T get(int index) {
        return items[index];
    }

    Run | Debug
    public static void main(String[] args) {
        var list = new GenericList<Integer>();
        list.add(item: 12);
        list.add(item: 15);
        System.out.println(list.get(index: 1));
    }
}
```

8.3 Code Reuse, Type Safety And Performance

The following code shows that we can put constraint on the generic class. Here it only accept “Number”, i.e Integer, Float, Double etc., other than that it will throw error during the compile time.

```
class GenericList2<T extends Number> {
    private T[] items = (T[]) new Object[10];
    private int count;

    public void add(T item) {
        items[count++] = item;
    }

    public T get(int index) {
        return items[index];
    }

    Run | Debug
    public static void main(String[] args) {
        var list = new GenericList2<String>();
    }
}
```

Error here

8.4 Collection And Generic

The following code demonstrates the build-in java.util.ArrayList library which use generic collection.

```
class SampleList {
    public void run() {
        List<String> listStrings = new ArrayList<String>();
        listStrings.add(e: "One");
        listStrings.add(e: "Two");
        listStrings.add(e: "Three");
        listStrings.add(e: "Four");
        System.out.println(listStrings);
    }

    Run | Debug
    public static void main(String[] args) {
        new SampleList().run();
    }
}
```

String Handling

- 9.1 Lower, Upper, Title Case function
- 9.2 Concat function
- 9.3 Substring function
- 9.4 Contain function
- 9.5 Replace function
- 9.6 IndexOf function

9.1 Lower and Upper Case function

Java String toLowerCase()

The java string toLowerCase() method returns the string in lowercase letter. In other words, it converts all characters of the string into lower case letter.

The java string toUpperCase() method returns the string in uppercase letter. In other words, it converts all characters of the string into lower upper letter.

Let's see an example:

```
String name = "AzMan";

// to lower case
System.out.println(name.toLowerCase());

// to upper case
System.out.println(name.toUpperCase());
```

9.2 Concat function

The Java String class concat() method combines specified string at the end of this string. It returns a combined string. It is like appending another string.

Signature

The signature of the string concat() method is given below:

```
public String concat (String anotherString)
```

Parameter

anotherString : another string i.e., to be combined at the end of this string.

Returns

combined string

Example

```
public void concat(String str) {
    String s1 = "java string";
    // The string s1 does not get changed, even though it is invoking the method
    // concat(), as it is immutable. Therefore, the explicit assignment is required
    // here.
    s1.concat(str: "is immutable");
    System.out.println(s1);
    s1 = s1.concat(str: " is immutable so assign it explicitly");
    System.out.println(s1);
}
```

Output

```
java string

java string is immutable so assign it explicitly
```

9.3 Substring function

Java String substring()

The Java String class substring() method returns a part of the string.

We pass beginIndex and endIndex number position in the Java substring method where beginIndex is inclusive, and endIndex is exclusive. In other words, the beginIndex starts from 0, whereas the endIndex starts from 1.

There are two types of substring methods in Java string.

Signature

```
public String substring(int startIndex)  // type - 1
```

and

```
public String substring(int startIndex, int endIndex)  // type - 2
```

If we don't specify endIndex, the method will return all the characters from startIndex.

Parameters

startIndex : starting index is inclusive

endIndex : ending index is exclusive

Returns

specified string

Exception Throws

StringIndexOutOfBoundsException is thrown when any one of the following conditions is met.

- if the start index is negative value
- end index is lower than starting index.
- Either starting or ending index is greater than the total number of characters present in the string.

Example

```
void substring() {
    String s1="javatpoint";
    System.out.println(s1.substring(beginIndex: 2,endIndex: 4));//returns va
    System.out.println(s1.substring(beginIndex: 2));//returns vatpoint
}
```

9.4 Contain function

Java String contains()

The Java String class contains() method searches the sequence of characters in this string. It returns true if the sequence of char values is found in this string otherwise returns false.

Signature

The signature of string contains() method is given below:

public boolean contains(CharSequence sequence)

Parameter

sequence: specifies the sequence of characters to be searched.

Returns

true if the sequence of char value exists, otherwise false.

Exception

NullPointerException : if the sequence is null.

Example

```
String str = "Hello Java readers";
boolean isContains = str.contains(s: "Java");
System.out.println(isContains);
// Case Sensitive
System.out.println(str.contains(s: "javatraining")); // false
```

Output

True

false

9.5 Replace function

The Java String class replace() method returns a string replacing all the old char or CharSequence to new char or CharSequence.

Since JDK 1.5, a new replace() method is introduced that allows us to replace a sequence of char values.

Signature

There are two types of replace() methods in Java String class.

- public String replace(char oldChar, char newChar)
- public String replace(CharSequence target, CharSequence replacement)

The second replace() method is added since JDK 1.5.

Parameters

- **oldChar**: old character
- **newChar**: new character
- **target**: target sequence of characters
- **replacement**: replacement sequence of characters

Returns

replaced string

Exception Throws

NullPointerException: if the replacement or target is equal to null.

Example

```
String s1 = "jomdemy.com is a very good website";
// replaces all occurrences of 'a' to 'e'
String replaceString = s1.replace(oldChar: 'a', newChar: 'e');
System.out.println(replaceString);
```

Output

Jomdemy.com is e very good website

9.6 IndexOf function

The **Java String class indexOf()** method returns the position of the first occurrence of the specified character or string in a specified string.

Signature

There are four overloaded indexOf() method in Java. The signature of indexOf() methods are given below:

No.	Method	Description
1	int indexOf(int ch)	It returns the index position for the given char value
2	int indexOf(int ch, int fromIndex)	It returns the index position for the given char value and from index
3	int indexOf(String substring)	It returns the index position for the given substring
4	int indexOf(String substring, int fromIndex)	It returns the index position for the given substring and from index

Parameters

ch: It is a character value, e.g. 'a'

fromIndex: The index position from where the index of the char value or substring is returned.

substring: A substring to be searched in this string.

Returns

Index of the searched string or character.

Example

```
String s1 = "this is index of example";
// passing substring
int index1 = s1.indexOf(str: "is");// returns the index of is substring
int index2 = s1.indexOf(str: "index");// returns the index of index substring
System.out.println(index1 + "   " + index2);// 2 8

// passing substring with from index
int index3 = s1.indexOf(str: "is", fromIndex: 4);// returns the index of is substring after 4th index
System.out.println(index3);// 5 i.e. the index of another is

// passing char value
int index4 = s1.indexOf(ch: 's');// returns the index of s char value
System.out.println(index4);// 3
```

Output

2

8

5

3

Image Manipulation

- 10.1 Loading Image
- 10.2 Editing Image
- 10.3 Download Image
- 10.4 ImageJ Library

10.1 Loading Image

Java BufferedImage class is a subclass of Image class. It is used to handle and manipulate the image data. A BufferedImage is made of ColorModel of image data. All BufferedImage objects have an upper left corner coordinate of (0, 0).

Constructors

This class supports three types of constructors.

The first constructor constructs a new BufferedImage with a specified ColorModel and Raster.

```
BufferedImage(ColorModel cm, WritableRaster raster,
boolean isRasterPremultiplied, Hashtable<?,?> properties)
```

The second constructor constructs a BufferedImage of one of the predefined image types.

```
BufferedImage(int width, int height, int imageType)
```

The third constructor constructs a BufferedImage of one of the predefined image types: TYPE_BYTE_BINARY or TYPE_BYTE_INDEXED.

```
BufferedImage(int width, int height, int imageType, IndexColorModel cm)
```

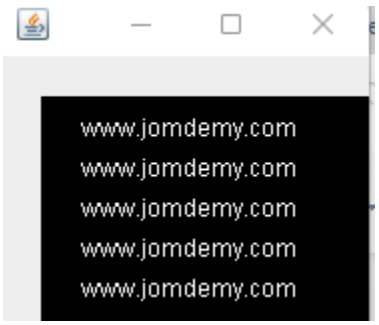
No	Method & Description
1	**copyData(WritableRaster outRaster)** It computes an arbitrary rectangular region of the BufferedImage and copies it into a specified WritableRaster.
2	**getColorModel()** It returns object of class ColorModel of an image.
3	**getData()** It returns the image as one large tile.
4	**getData(Rectangle rect)** It computes and returns an arbitrary region of the BufferedImage.
5	**getGraphics()** This method returns a Graphics2D, retains backwards compatibility.
6	**getHeight()** It returns the height of the BufferedImage.
7	**getMinX()** It returns the minimum x coordinate of this BufferedImage.
8	**getMinY()** It returns the minimum y coordinate of this BufferedImage.
9	**getRGB(int x, int y)** It returns an integer pixel in the default RGB color model (TYPE_INT_ARGB) and default sRGB colorspace.
10	**getType()** It returns the image type.

10.2 Editing Image

The following example demonstrates the use of java **BufferedImage** class that draw some text on the screen using Graphics Object:

```
import java.awt.Graphics;
import java.awt.Image;
import java.awt.image.BufferedImage;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class Test extends JPanel {

   public void paint(Graphics g) {
      Image img = createImageWithText();
      g.drawImage(img, x: 20,y: 20,this);
   }

   private Image createImageWithText() {
      BufferedImage bufferedImage = new BufferedImage(width: 200,height: 200,BufferedImage.TYPE_INT_RGB);
      Graphics g = bufferedImage.getGraphics();

      g.drawString(str: "www.jomdemy.com", x: 20,y: 20);
      g.drawString(str: "www.jomdemy.com", x: 20,y: 40);
      g.drawString(str: "www.jomdemy.com", x: 20,y: 60);
      g.drawString(str: "www.jomdemy.com", x: 20,y: 80);
      g.drawString(str: "www.jomdemy.com", x: 20,y: 100);

      return bufferedImage;
   }

   Run | Debug
   public static void main(String[] args) {
      JFrame frame = new JFrame();
      frame.getContentPane().add(new Test());

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(width: 200, height: 200);
      frame.setVisible(b: true);
   }
}
```

Output



10.3 Download Image

The following example demonstrates the use of java URL class to download an image from the internet.

```
import java.net.URL;

public class Download {

    Run | Debug
    public static void main(String[] args) throws Exception {

        try{
            String fileName = "java1.png";
            String website = "http://jomdemy.com/images/"+fileName;

            System.out.println("Downloading File From: " + website);

            URL url = new URL(website);
            InputStream inputStream = url.openStream();
            OutputStream outputStream = new FileOutputStream(fileName);
            byte[] buffer = new byte[2048];

            int length = 0;

            while ((length = inputStream.read(buffer)) != -1) {
                System.out.println("Buffer Read of length: " + length);
                outputStream.write(buffer, off: 0, length);
            }

            inputStream.close();
            outputStream.close();

        } catch(Exception e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

Output

```
Downloading File From: http://jomdemy.com/images/java1.png
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 1794
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 64
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 2048
Buffer Read of length: 955
```

10.4 ImageJ Library

ImageJ is a Java-based software created for working with images. It has quite a lot of plugins, available here. We will be using API only, as we want to perform processing by ourselves.

It is quite a powerful library, better than Swing and AWT, as it's creation purpose was image processing and not GUI operations. Plugins contain many free to use algorithms, which is a good thing when we want to learn image processing and quickly see the results, rather than solve math and optimization problems laying under IP algorithms.

To start working with ImageJ, simply add a dependency to your project's pom.xml file:

```
<dependency>
  <groupId>net.imagej</groupId>
  <artifactId>ij</artifactId>
  <version>1.51h</version>
</dependency>
```

To load the image, you need to use the openImage() static method, from IJ class:

```
ImagePlus imp = IJ.openImage("path/to/your/image.jpg");
```

Full Example

```
// draw a rectangle on an image and save it into a file
public void drawRectangel() {
    ImagePlus imp = IJ.openImage(path: "assets/test.jpeg");
    ImageProcessor ip = imp.getProcessor();
    // ip.invert();
    ip.setColor(Color.red);
    ip.setLineWidth(width: 4);
    ip.drawRect(x: 50, y: 50, imp.getWidth() - 100, imp.getHeight() - 100);
    // imp.show(); // show popup window of the image
    FileSaver fs = new FileSaver(imp);
    fs.saveAsJpeg(path: "assets/test2.jpeg");
}
```

Regular Expression

11.1 What Is RE

A regular expression is a *sequence of characters that forms a search pattern*. When you search for data in a text, you can use this search pattern to describe what you are searching for. A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of text search and text replace operations. Java does not have a built-in Regular Expression class, but we can import the java.util.regex package to work with regular expressions. The package includes the following classes:

- Pattern Class - Defines a pattern (to be used in a search)
- Matcher Class - Used to search for the pattern
- PatternSyntaxException Class - Indicates syntax error in a regular expression pattern

11.2 Define Pattern

It is the compiled version of a regular expression. It is used to define a pattern for the regex engine.

No.	Method	Description
1	static Pattern **compile**(String regex)	compiles the given regex and returns the instance of the Pattern.
2	Matcher **matcher**(CharSequence input)	creates a matcher that matches the given input with the pattern.
3	static boolean **matches**(String regex, CharSequence input)	It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern.
4	String[] **split**(CharSequence input)	splits the given input string around matches of given pattern.
5	String **pattern**()	returns the regex pattern.

In this example, The word "schools" is being searched for in a sentence. First, the pattern is created using the Pattern.compile() method. The first parameter indicates which pattern is being searched for and the second parameter has a flag to indicates that the search should be case-insensitive. The second parameter is optional.

The matcher() method is used to search for the pattern in a string. It returns a Matcher object which contains information about the search that was performed. The find() method returns true if the pattern was found in the string and false if it was not found.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Sample1 {
    Run | Debug
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile(regex: "school", Pattern.CASE_INSENSITIVE);
        Matcher matcher = pattern.matcher(input: "Visit my school!");
        boolean matchFound = matcher.find();

        if (matchFound) {
            System.out.println(x: "Match found");
        } else {
            System.out.println(x: "Match not found");
        }
    }
}
```

The result : Match Found

Flags in the compile() method change how the search is performed. Here are a few of them:

- Pattern.CASE_INSENSITIVE - The case of letters will be ignored when performing a search.
- Pattern.LITERAL - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters when performing a search.
- Pattern.UNICODE_CASE - Use it together with the CASE_INSENSITIVE flag to also ignore the case of letters outside of the English alphabet

Regular Expression Pattern

The first parameter of the Pattern.compile() method is the pattern. It describes what is being searched for. Brackets are used to find a range of characters:

Expression	Description
[abc]	Find one character from the options between the brackets
[^abc]	Find one character NOT between the brackets
[0-9]	Find one character from the range 0 to 9

Metacharacters

Metacharacters are characters with a special meaning:

Metacharacter	Description
|	Find a match for any one of the patterns separated by | as in: cat|dog|fish
.	Find just one instance of any character
^	Finds a match as the beginning of a string as in: ^Hello
$	Finds a match at the end of the string as in: World$
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx

Quantifiers

Quantifiers define quantities:

Quantifier	Description
n+	Matches any string that contains at least one n
n*	Matches any string that contains zero or more occurrences of n
n?	Matches any string that contains zero or one occurrences of n
n{x}	Matches any string that contains a sequence of X n's
n{x,y}	Matches any string that contains a sequence of X to Y n's
n{x,}	Matches any string that contains a sequence of at least X n's

11.3 Define Matcher

It implements the MatchResult interface. It is a regex engine which is used to perform match operations on a character sequence.

No.	Method	Description
1	boolean **matches**()	test whether the regular expression matches the pattern.
2	boolean **find**()	finds the next expression that matches the pattern.
3	boolean **find**(int start)	finds the next expression that matches the pattern from the given start number.
4	String **group**()	returns the matched subsequence.

5	int **start**()	returns the starting index of the matched subsequence.
6	int **end**()	returns the ending index of the matched subsequence.
7	int **groupCount**()	returns the total number of the matched subsequence.

Using Java Bean

- 13.1 Reusable Component
- 13.2 Properties, Set, Get

13.1 Reusable Component

A JavaBean is a Java class that should follow the following conventions:

- It should have a no-arg constructor.
- It should be Serializable.
- It should provide methods to set and get the values of the properties, known as getter and setter methods.

Why use JavaBean?

According to Java white paper, it is a reusable software component. A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides easy maintenance.

Simple example of JavaBean class

```
public class Employee implements java.io.Serializable {
    private int id;
    private String name;

    public Employee() {
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

How to access the JavaBean class?

To access the JavaBean class, we should use getter and setter methods.

```
class Test {
    Run | Debug
    public static void main(String args[]) {
        Employee e = new Employee();// object is created
        e.setName(name: "Azman");// setting value to the object
        System.out.println(e.getName());
    }
}
```

13.2 Properties, Set, Get

A JavaBean property is a named feature that can be accessed by the user of the object. The feature can be of any Java data type, containing the classes that you define.

A JavaBean property may be read, write, read-only, or write-only. JavaBean features are accessed through two methods in the JavaBean's implementation class:

1. getPropertyName ()

For example, if the property name is firstName, the method name would be getFirstName() to read that property. This method is called the accessor.

2. setPropertyName ()

For example, if the property name is firstName, the method name would be setFirstName() to write that property. This method is called the mutator.

Advantages of JavaBean

The following are the advantages of JavaBean:/p>

The JavaBean properties and methods can be exposed to another application.

It provides an easiness to reuse the software components.

Disadvantages of JavaBean

The following are the disadvantages of JavaBean:

JavaBeans are mutable. So, it can't take advantages of immutable objects.

Creating the setter and getter method for each property separately may lead to the boilerplate code.

Accessing JavaBeans

The useBean action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows −

```
<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>
```

Here values for the scope attribute can be a page, request, session or application based on your requirement. The value of the id attribute may be any value as a long as it is a unique name among other useBean declarations in the same JSP.

Following example shows how to use the useBean action:

```
<html>
   <head>
      <title>useBean Example</title>
   </head>

   <body>
      <jsp:useBean id = "date" class = "java.util.Date" />
      <p>The date/time is <%= date %>
   </body>
</html>
```

Networking and Socket Programming

14.1 Socket Class

Java Socket programming is used for communication between the applications running on different JRE. Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

- IP Address of Server, and
- Port number.

Here, we are going to make one-way client and server communication. In this application, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The Socket class is used to communicate client and server. Through this class, we can read and write message. The ServerSocket class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.

1. IP Address of Server, and
2. Port number.

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

Important methods

Method	Description
1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2) public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public synchronized void close()	closes this socket

14.2 Serversocket Class

The **ServerSocket** class can be used to create a server socket. This object is used to establish communication with the clients.

Important methods

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Example of Java Socket Programming

Creating Server:

To create the server application, we need to create the instance of ServerSocket class. Here, we are using 6666 port number for the communication between the client and server. You may also choose any other port number. The accept() method waits for the client. If clients connects with the given port number, it returns an instance of Socket.

```
ServerSocket ss = new ServerSocket(6666);

Socket s = ss.accept();//establishes connection and waits for the client
```

Creating Client:

To create the client application, we need to create the instance of Socket class. Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

```
Socket s=new Socket("localhost",6666);
```

14.3 Creating A Server

Let's see a simple of Java socket programming where client sends a text and server receives and prints it.

```
public class MyServer {
    Run | Debug
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(port: 6666);
            Socket s = ss.accept();// establishes connection
            DataInputStream dis = new DataInputStream(s.getInputStream());
            String str = (String) dis.readUTF();
            System.out.println("message= " + str);
            ss.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

14.4 Creating A Client

Here is the client socket

```java
public class MyClient {
    Run | Debug
    public static void main(String[] args) {
        try {
            Socket s = new Socket(host: "localhost", port: 6666);
            DataOutputStream dout = new DataOutputStream(s.getOutputStream());
            dout.writeUTF(str: "Hello Server");
            dout.flush();
            dout.close();
            s.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Database Connectivity (JDBC)

15.1 Why JDBC

JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records

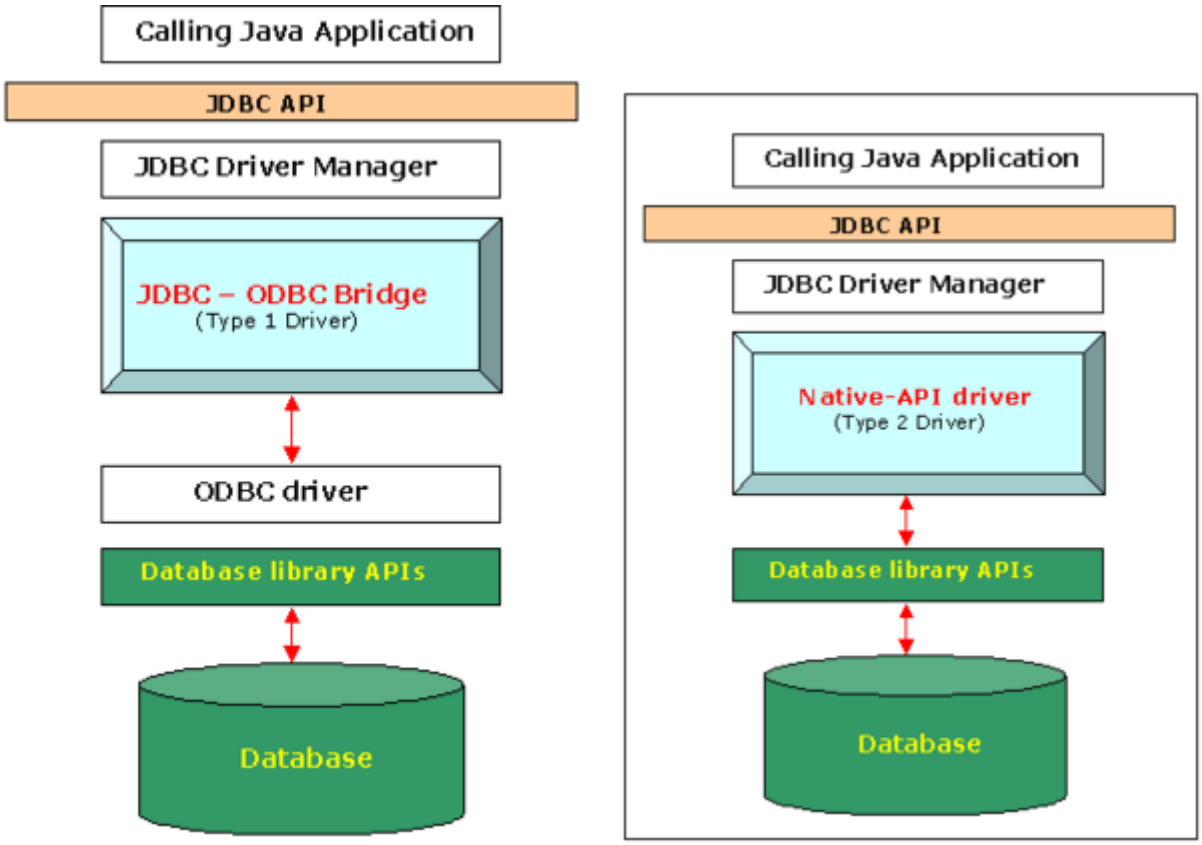
15.2 JDBC Drivers

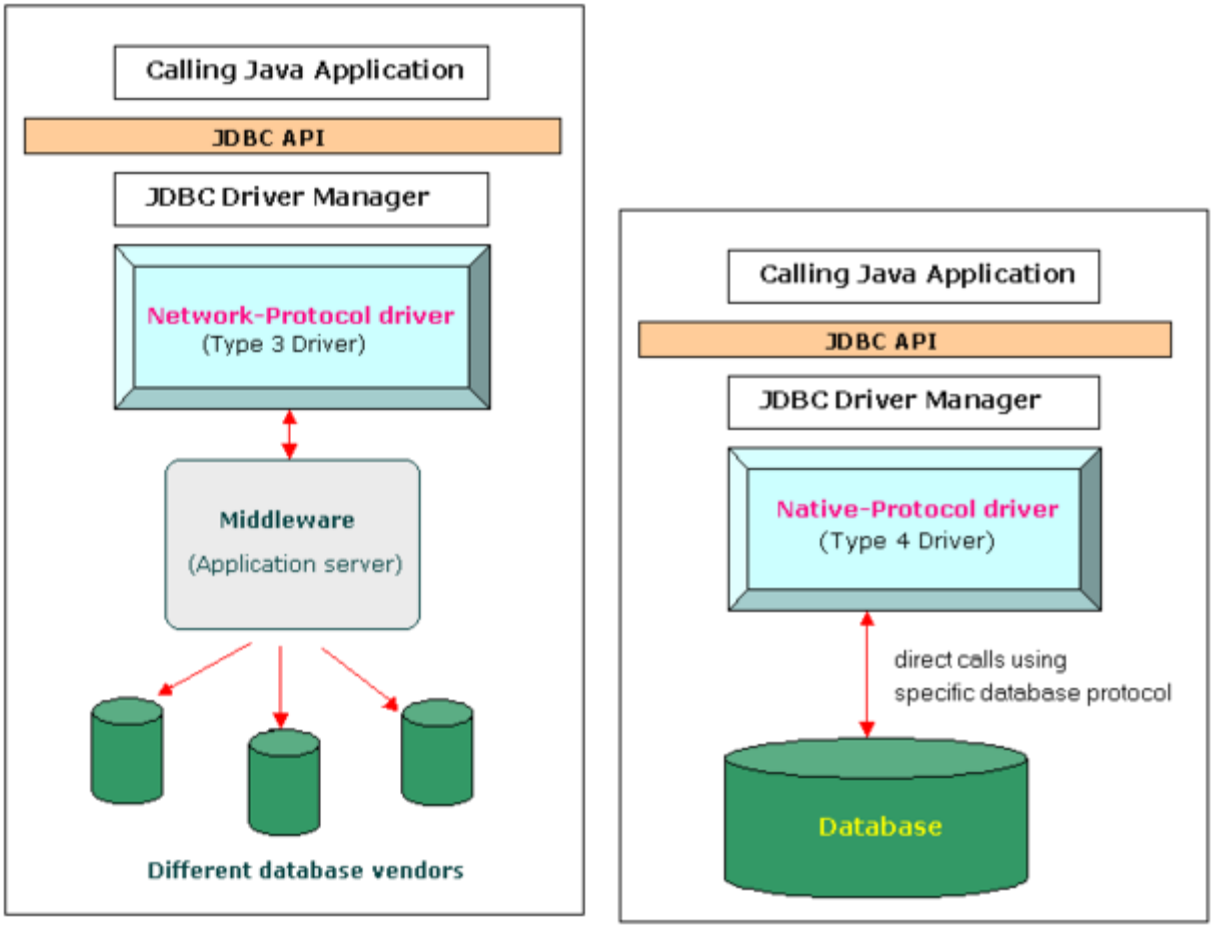
A JDBC driver is a software component enabling a Java application to interact with a database. JDBC drivers are analogous to ODBC drivers, ADO.NET data providers, and OLE DB providers.

JDBC technology drivers fit into one of four categories.

- JDBC-ODBC bridge (type 1 driver)
- Native-API driver (type 2 driver)
- Network-Protocol driver (Middleware driver) (type 3 driver)

- Database-Protocol driver (Pure Java driver) or thin driver (type 4 driver)





To connect with individual databases, JDBC (the Java Database Connectivity API) requires drivers for each database. The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database.

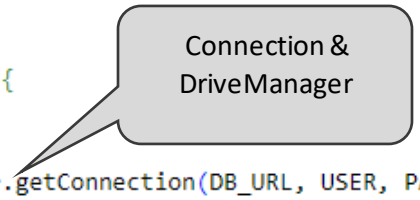
In order to use the JDBC driver, we need to get it using Maven dependency as in the following configuration of pom.xml.

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.29</version>
</dependency>
```

15.3 Connection Interface

This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.

```
import java.sql.*;

public class Sample1 {
    static final String DB_URL = "jdbc:mysql://localhost/jompos";
    static final String USER = "root";
    static final String PASS = "";
    static final String QUERY = "SELECT * FROM user";

    Run | Debug
    public static void main(String[] args) {
        // Open a connection
        try {
            Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(QUERY);
```



15.4 DriverManager Class

This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

15.5 Statement Interface

You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

15.6 Resultset Interface

These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

15.7 Preparedstatement Interface

You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures

15.8 ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

15.9 Storing Data

The following code show how to insert data into MySQL database.

```
import java.sql.*;

public class StoringData {
    static final String DB_URL = "jdbc:mysql://localhost/jompos";
    static final String USER   = "root";
    static final String PASS   = "";
    static final String QUERY  = "INSERT INTO user (user_id, name, email) VALUES (?, ?, ?)";

    Run | Debug
    public static void main(String[] args) {
        PreparedStatement pstmt = null;
        Connection conn = null;

        try {
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            pstmt = conn.prepareStatement(QUERY);
            pstmt.setString(parameterIndex: 1, x: "abu");
            pstmt.setString(parameterIndex: 2, x: "Abu Hassan");
            pstmt.setString(parameterIndex: 3, x: "abu@gmail.com");
            pstmt.executeUpdate();
            pstmt.close();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

15.10 Retrieving Data

The following code demonstrate how to retrieve data using prepareStatement.

```
import java.sql.*;

public class Sample2 {
    static final String DB_URL = "jdbc:mysql://localhost/jompos";
    static final String USER   = "root";
    static final String PASS   = "";
    static final String QUERY  = "SELECT * FROM user WHERE id = ?";

    Run | Debug
    public static void main(String[] args) {
        PreparedStatement pstmt = null;
        Connection conn = null;

        try {
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            pstmt = conn.prepareStatement(QUERY);
            pstmt.setInt(parameterIndex: 1, x: 1);
            ResultSet rs = pstmt.executeQuery();
            while (rs.next()) {
                System.out.print("ID: " + rs.getInt(columnLabel: "id"));
                System.out.print(", User ID: " + rs.getString(columnLabel: "user_id"));
                System.out.print(", Name: " + rs.getString(columnLabel: "name"));
                System.out.println(", Email: " + rs.getString(columnLabel: "email"));
            }
            pstmt.close();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Project Management with Maven

16.1 Purpose

Maven is a build tool that helps in project management. The tool help in building and documenting the project. Maven is written in Java and is used Java, C#, Ruby, and Scala project. It'd based on the Project Object Model (POM).

Project Object Model is an XML file that has all the information regarding project and configuration details. When we tend to execute a task, Maven searches for the POM in the current directory.

The tool is used to **build** and manage any Java-based project. It simplifies the day-to-day work of Java Developers and helps then in their projects. It helps in downloading **dependencies**, which refer to the libraries or JAR files.

- Getting the right JAR files for each project as there may be different version of separate package.

- To download dependencies visiting of the official website of different software is not needed. We can just visit "mvnrepository.com"

- Help to create the right project structure which is essential for execution.

Build tool is essential for the purpose of building. It's needed for the following processes:

- Generating source code
- Generating documentation from the source code
- Compiling the source code
- Packaging of the complied codes into JAR files
- Installing the packaged code in local repository, server, or central repository

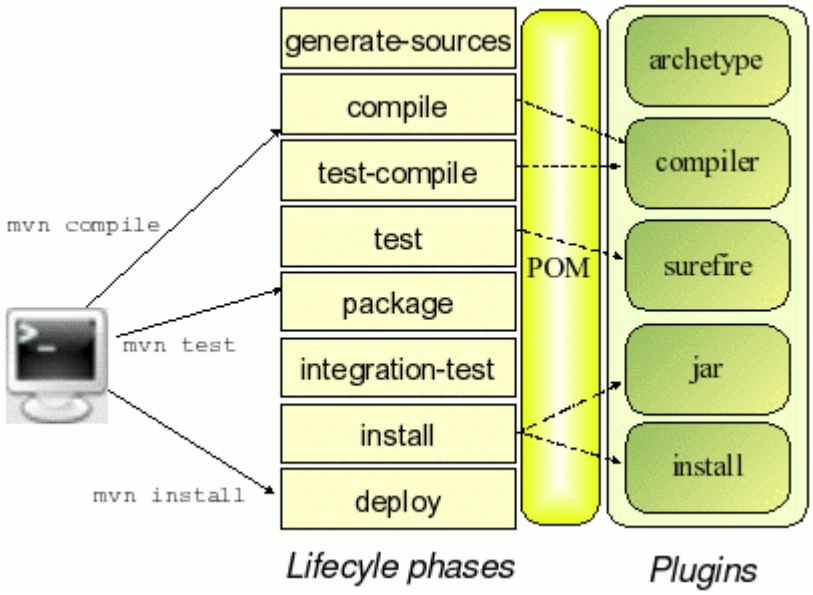
Maven Characteristics

- IDE agnostic
- IDE supported

- Declarative dependency management
- Plugin based architecture

Maven Life Cycles, Phases and Goals

- Build lifecycle consist of a sequence of build phases, and each build phase consists of a sequence of goals.
- Each goal is responsible for a particular task
- When phase is run all the goals related to that phase and its plugins are also compiled



Build Plugins

- Maven plugin refer to the group of goals. These goals may or may not be of the same phase.
- The plugins are used to perform specific goals.
- Maven has its own standard plugins that can be used. If wanted, we can also implement our own in Java.

16.2 Easy Project Setup

We need to follow the following steps to create our first maven project.

- Download maven and then extract it
- Set environment variables for JAVA_HOME and M2_HOME
- Check installation by typing >mvn –version

The following command generate a Java Maven project.

>mvn archetype: generate

-Darchetype**GroupId**=org.apache.maven.archetypes

-Darchetype**ArtifactId**=maven-archetype-quickstart

-Darchetype**Version**=1.4

16.3 Build Project (Jar / War)

>mvn clean install

The command will delete previous compiled and packaged files, compile *.java into .class and generate .war / .jar file.

16.4 Dependency Management

To add dependency, we add dependency section as in the following config in pom.xml. Normally we copy the setting from mvnrepository.com. All jar files related to the dependencies will be automatically downloaded.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>2bytes.com</groupId>
    <artifactId>sakila-springboot</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>sakila-springboot</name>
    <description>A reference project for Spring Boot</description>
    <properties>
        <java.version>11</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        ...
```

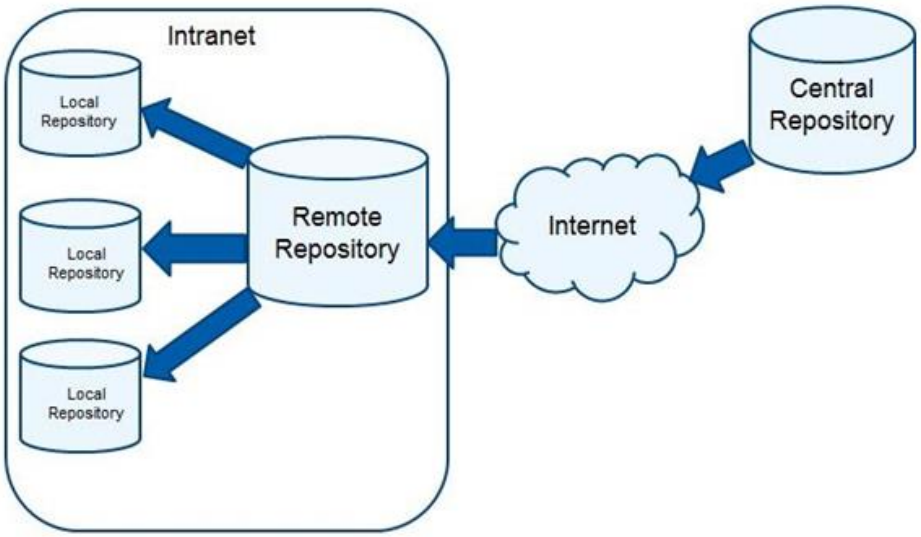
16.5 Repository

There are 3 types of Maven repositories:

1. Local repo
    - see C:\Users\<user name>\.m2\repository
    - Reside on developer machine

2. Remote repo
    - A server within an organization

3. Central repo
    - https://mvnrepository.com

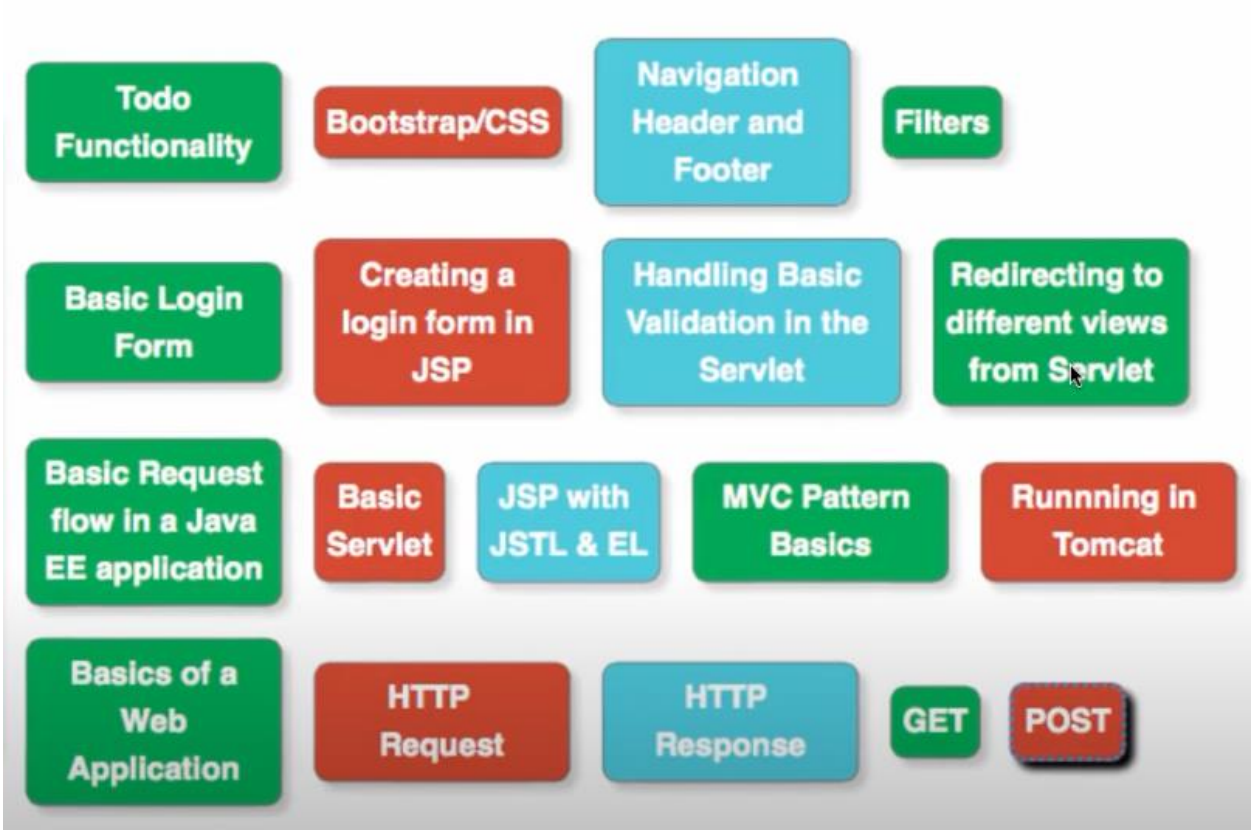

Intranet
Local Repository
Local Repository
Local Repository
Remote Repository
Internet
Central Repository

Web and Servlet

- 17.1 What Is Servlet
- 17.2 Servlet Api
- 17.3 Servlet Lifecycle
- 17.4 Servlet Project Setup
- 17.5 Servlet Context And Servlet Config
- 17.6 Session
- 17.7 Cookie
- 17.8 Request
- 17.9 Response

Introduction

The following diagram depict what to cover under this topic. This topic is part of Java Enterprise Edition or JEE.



We will use the following technologies and tools in order to develop the JEE application.

- JSP 2.2
- Eclipse 2022-03

- JDK 1.8 or later
- Apache Tomcat 9.x
- JSTL 1.2.3
- Servlet API 4.x
- JSP 2.3
- MySQL connector jar

In the incoming example, we will use MVC methodology as the development of example program.

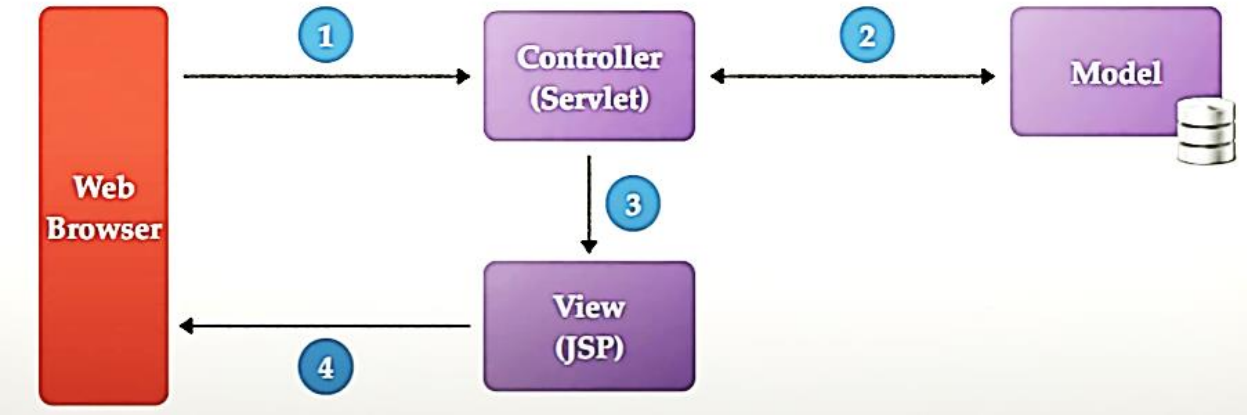
MVC Overview:

- Model-View-Controller (MVC) is a pattern used in software engineering to separate application logic from the user interface. As the name implies, the MVC pattern has three layers.

- The Model defines the business layer of the application, the Controller manages the flow of the application, and the View defines the presentation layer of the application.

Here're some key features of the pattern:

- It separate the presentation layer from the business layer
- The Controller performs the action of invoking the Model and sending data to view
- The Model is not even aware that it is used by some web application or a desktop application



17.1 What Is Servlet

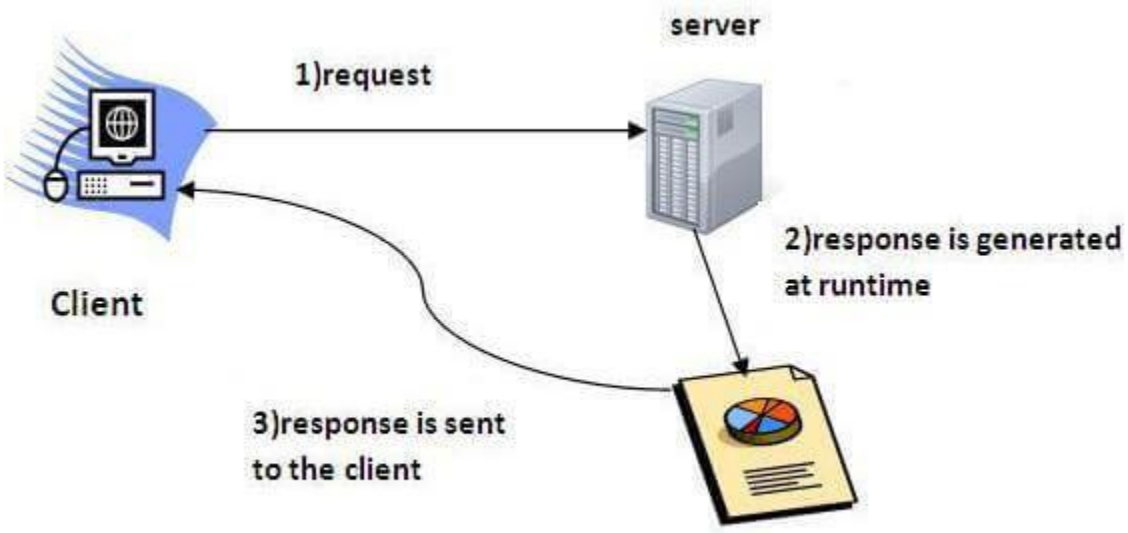
Servlet technology is used to create a web application (resides at server side and generates a dynamic web page).

Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below.

There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

Servlet can be described in many ways, depending on the context.

- Servlet is a technology which is used to create a web application.
- Servlet is an API that provides many interfaces and classes including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page



17.2 Servlet API

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.

The javax.servlet package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The javax.servlet.http package contains interfaces and classes that are responsible for http requests only.

Let's see what are the interfaces of javax.servlet package.

Interfaces in javax.servlet package

There are many interfaces in javax.servlet package. They are as follows:

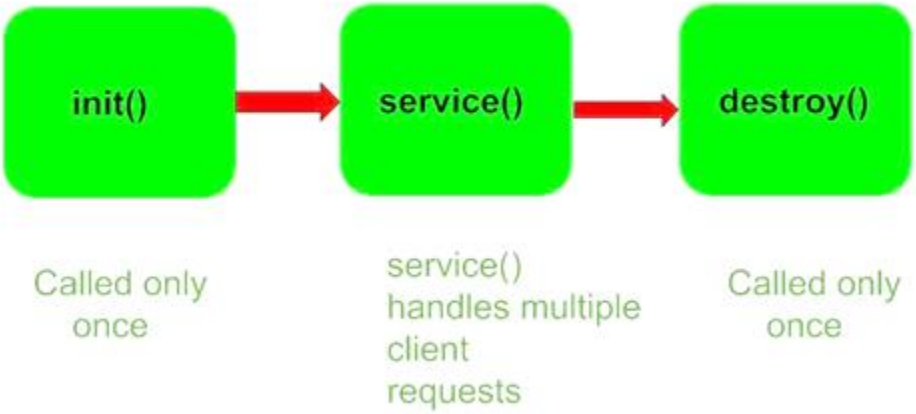
- Servlet
- ServletRequest
- ServletResponse
- RequestDispatcher
- ServletConfig
- ServletContext
- SingleThreadModel
- Filter
- FilterConfig
- FilterChain
- ServletRequestListener
- ServletRequestAttributeListener
- ServletContextListener
- ServletContextAttributeListener

17.3 Servlet Lifecycle

The following describe the servlet lifecycle. The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

- Servlet class is loaded.
- Servlet instance is created.

- init method is invoked.
- service method is invoked.
- destroy method is invoked.
-



As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below:

```
public void init(ServletConfig config) throws ServletException
```

4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

```
public void service(ServletRequest request, ServletResponse response)
  throws ServletException, IOException
```

5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

```
public void destroy()
```

17.4 Servlet Project Setup

We will follow the following stepas in order to create a servlet project.

- Create IntelliJ Idea Dynamic web project
- Add dependencies
- Project structure
- MySQL database setup
- Create a JavaBean (model) – User.java
- Create a UserService.java
- Create a UserController.java
- Creae user listing JSP page – user-list.jsp
- Create a user form JSP page – user-form.jsp
- Create error JSP page
- Deploying and testing the Application Demo

17.5 Servlet Context and Servlet Config

An object of **ServletContext** is created by the web container at time of deploying the project. This object can be used to get configuration information from **web.xml** file. There is only one **ServletContext** object per web application.

If any information is shared to many servlet, it is better to provide it from the **web.xml** file using the <context-param> element.

Advantage of ServletContext

Easy to maintain if any information is shared to all the servlet, it is better to make it available for all the servlet. We provide this information from the **web.xml** file, so if the information is changed, we don't need to modify the servlet. Thus, it removes maintenance problem.

Usage of ServletContext Interface

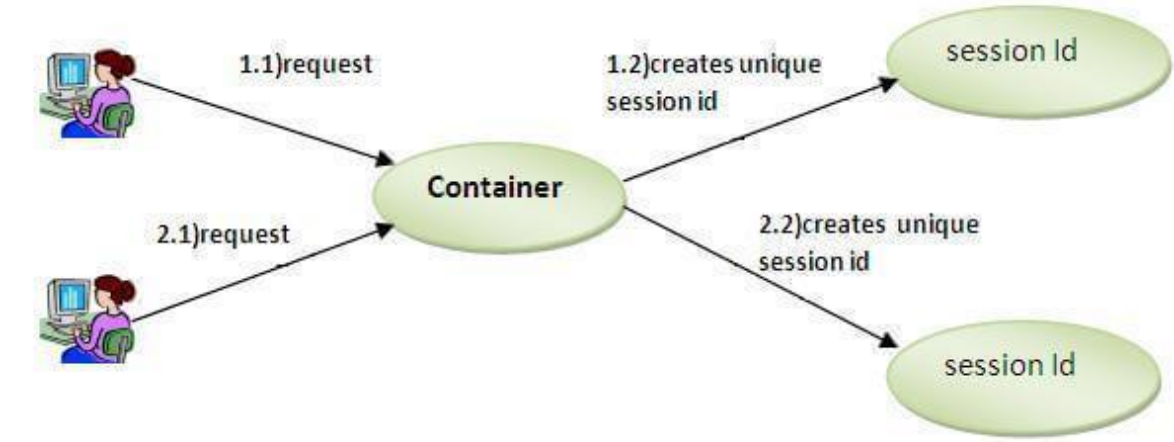
There can be a lot of usage of **ServletContext** object. Some of them are as follows:

- The object of **ServletContext** provides an interface between the container and servlet.
- The **ServletContext** object can be used to get configuration information from the web.xml file.
- The **ServletContext** object can be used to set, get or remove attribute from the web.xml file.
- The **ServletContext** object can be used to provide inter-application communication.

17.6 Session

In such case, container creates a session id for each user.The container uses this id to identify the particular user.An object of HttpSession can be used to perform two tasks:

- bind objects
- view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.



How to get the HttpSession object?

The HttpServletRequest interface provides two methods to get the object of HttpSession:

- public HttpSession getSession (): Returns the current session associated with this request, or if the request does not have a session, creates one.
- public HttpSession getSession (boolean create): Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

Commonly used methods of HttpSession interface

- public String getId (): Returns a string containing the unique identifier value.
- public long getCreationTime (): Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970, GMT.
- public long getLastAccessedTime (): Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970, GMT.
- public void invalidate (): Invalidates this session then unbinds any objects bound to it.

Example of using HttpSession

```
<form action="servlet1">
  Name:<input type="text" name="userName" /><br />
  <input type="submit" value="go" />
</form>
```

```
import java.io.*;
import javax.servlet.http.*;

public class FirstServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response) {
        try {

            response.setContentType("text/html");
            PrintWriter out = response.getWriter();

            String n = request.getParameter("userName");
            out.print("Welcome " + n);

            HttpSession session = request.getSession();
            session.setAttribute("uname", n);

            out.print(s: "<a href='servlet2'>visit</a>");

            out.close();

        } catch (Exception e) {
            System.out.println(e);
        }
    }

}
```

Session handling

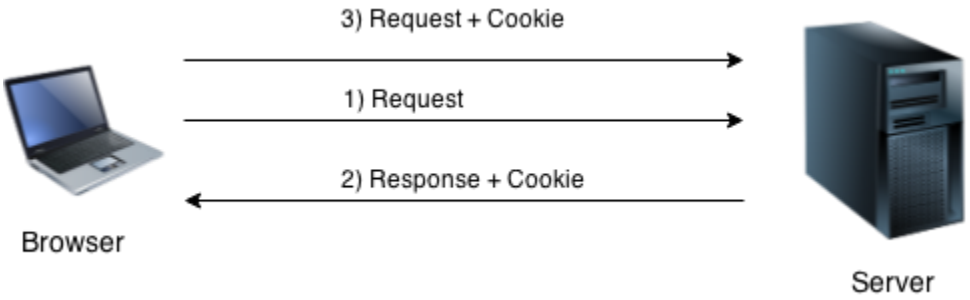
```java
import java.io.*;
import javax.servlet.http.*;

public class SecondServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response) {
        try {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();

            HttpSession session = request.getSession(false);
            String n = (String) session.getAttribute("uname");
            out.print("Hello " + n);

            out.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }

}
```

Web.xml

```xml
<web-app>

    <servlet>
        <servlet-name>s1</servlet-name>
        <servlet-class>FirstServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>s1</servlet-name>
        <url-pattern>/servlet1</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>s2</servlet-name>
        <servlet-class>SecondServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>s2</servlet-name>
        <url-pattern>/servlet2</url-pattern>
    </servlet-mapping>

</web-app>
```

17.7 Cookie

A cookie is a small piece of information that is persisted between the multiple client requests. A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.



There are 2 types of cookies in servlets.

- Non-persistent cookie
- Persistent cookie

Non-persistent cookie

It is valid for single session only. It is removed each time when user closes the browser.

Persistent cookie

It is valid for multiple session. It is not removed each time when user closes the browser. It is removed only if user logout or sign-out.

Advantage of Cookies

- Simplest technique of maintaining the state.
- Cookies are maintained at client side.

Disadvantage of Cookies

- It will not work if cookie is disabled from the browser.
- Only textual information can be set in Cookie object.

How to create Cookie?

Let's see the simple code to create cookie.

```
// create cookie
Cookie ck = new Cookie("user","John Doe");//creating cookie object
response.addCookie(ck);//adding cookie in the response
```

How to delete Cookie?

Let's see the simple code to delete cookie. It is mainly used to logout or sign-out the user.

```
// delete cookie
Cookie ck2 = new Cookie("user","");//deleting value of cookie
ck2.setMaxAge(0);//changing the maximum age to 0 seconds
response.addCookie(ck2);//adding cookie in the response
```

How to get Cookies?

Let's see the simple code to get all the cookies.

```
PrintWriter out = response.getWriter();
Cookie ck3[] = request.getCookies();
for(int i = 0; i < ck3.length; i++){
    out.print("<br>"+ck3[i].getName()+" "+ck3[i].getValue());//printing name and value of cookie
}
```

17.8 Request

An object of **ServletRequest** is used to provide the client request information to a servlet such as content type, content length, parameter names and values, header information, attributes etc.

Methods of ServletRequest interface

There are many methods defined in the ServletRequest interface. Some of them are as follows:

Method	Description
public String getParameter(String name)	is used to obtain the value of a parameter by name.

public String[] getParameterValues(String name)	returns an array of String containing all values of given parameter name. It is mainly used to obtain values of a Multi select list box.
java.util.Enumeration getParameterNames()	returns an enumeration of all of the request parameter names.
public int getContentLength()	Returns the size of the request entity data, or -1 if not known.
public String getCharacterEncoding()	Returns the character set encoding for the input of this request.
public String getContentType()	Returns the Internet Media Type of the request entity data, or null if not known.
public ServletInputStream getInputStream() throws IOException	Returns an input stream for reading binary data in the request body.
public abstract String getServerName()	Returns the host name of the server that received the request.
public int getServerPort()	Returns the port number on which this request was received.

Example of ServletRequest to display the name of the user

```
<form action="welcome" method="get">
  Enter your name<input type="text" name="name" /><br />
  <input type="submit" value="login" />
</form>

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;

public class DemoServ extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter pw = res.getWriter();

        String name = req.getParameter("name");// will return value
        pw.println("Welcome " + name);

        pw.close();
    }
}
```

17.9 Response

There are following methods which can be used to set HTTP response header in your servlet program. These methods are available with **HttpServletResponse** object.

No.	Method & Description
1	**String encodeRedirectURL(String url)** Encodes the specified URL for use in the sendRedirect method or, if encoding is not needed, returns the URL unchanged.
2	**String encodeURL(String url)** Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.

3	**boolean containsHeader(String name)** Returns a Boolean indicating whether the named response header has already been set.
4	**boolean isCommitted()** Returns a Boolean indicating if the response has been committed.
5	**void addCookie(Cookie cookie)** Adds the specified cookie to the response.
6	**void addDateHeader(String name, long date)** Adds a response header with the given name and date-value.
7	**void addHeader(String name, String value)** Adds a response header with the given name and value.
8	**void addIntHeader(String name, int value)** Adds a response header with the given name and integer value.
9	**void flushBuffer()** Forces any content in the buffer to be written to the client.
10	**void reset()** Clears any data that exists in the buffer as well as the status code and headers.
11	**void resetBuffer()** Clears the content of the underlying buffer in the response without clearing headers or status code.
12	**void sendError(int sc)** Sends an error response to the client using the specified status code and clearing the buffer.
13	**void sendError(int sc, String msg)**

	Sends an error response to the client using the specified status.
14	**void sendRedirect(String location)** Sends a temporary redirect response to the client using the specified redirect location URL.
15	**void setBufferSize(int size)** Sets the preferred buffer size for the body of the response.
16	**void setCharacterEncoding(String charset)** Sets the character encoding (MIME charset) of the response being sent to the client, for example, to UTF-8.
17	**void setContentLength(int len)** Sets the length of the content body in the response In HTTP servlets, this method sets the HTTP Content-Length header.
18	**void setContentType(String type)** Sets the content type of the response being sent to the client, if the response has not been committed yet.
19	**void setDateHeader(String name, long date)** Sets a response header with the given name and date-value.
20	**void setHeader(String name, String value)** Sets a response header with the given name and value.
21	**void setIntHeader(String name, int value)** Sets a response header with the given name and integer value
22	**void setLocale(Locale loc)** Sets the locale of the response, if the response has not been committed yet.

23	**void setStatus(int sc)** Sets the status code for this response

Code example of servlet response

```java
// Import required java libraries
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// Extend HttpServlet class
public class Refresh extends HttpServlet {

  // Method to handle GET method request.
  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    // Set refresh, autoload time as 5 seconds
    response.setIntHeader("Refresh", 5);

    // Set response content type
    response.setContentType("text/html");

    // Get current time
    Calendar calendar = new GregorianCalendar();
    String am_pm;
    int hour = calendar.get(Calendar.HOUR);
    int minute = calendar.get(Calendar.MINUTE);
    int second = calendar.get(Calendar.SECOND);

    if(calendar.get(Calendar.AM_PM) == 0)
      am_pm = "AM";
    else
      am_pm = "PM";

    String CT = hour+":"+ minute +":"+ second +" "+ am_pm;

    PrintWriter out = response.getWriter();
    String title = "Auto Refresh Header Setting";
    String docType =
      "<!doctype html public \"-//w3c//dtd html 4.0 " + "transitional//en\">\n";

    out.println(docType +
      "<html>\n" +
      "<head><title>" + title + "</title></head>\n"+
      "<body bgcolor = \"#f0f0f0\">\n" +
      "<h1 align = \"center\">" + title + "</h1>\n" +
      "<p>Current Time is: " + CT + "</p>\n"
    );
  }

  // Method to handle POST method request.
  public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    doGet(request, response);
  }
}
```

Web and JSP

18.1 What Is JSP

Java Server Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications. JSP have access to the entire family of Java APIs, including the JDBC API to access enterprise databases. This tutorial will teach you how to use Java Server Pages to develop your web applications in simple and easy steps.

Why to Learn JSP?

- JavaServer Pages often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But JSP offers several advantages in comparison with the CGI.

- Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself instead of having separate CGI files.

- JSP are always compiled before they are processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested.

- JavaServer Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including JDBC, JNDI, EJB, JAXP, etc.

- JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

Finally, JSP is an integral part of Java EE, a complete platform for enterprise class applications. This means that JSP can play a part in the simplest applications to the most complex and demanding.

18.2 JSP API

The JSP API consists of two packages:

- javax.servlet.jsp
- javax.servlet.jsp.tagext

javax.servlet.jsp package

The javax.servlet.jsp package has two interfaces and classes. The two interfaces are as follows:

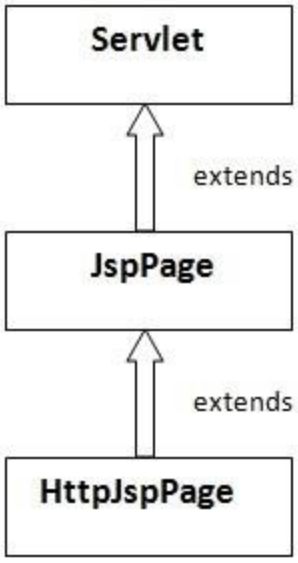
- JspPage
- HttpJspPage

The classes are as follows:

- JspWriter
- PageContext
- JspFactory
- JspEngineInfo
- JspException
- JspError

The JspPage interface

According to the JSP specification, all the generated servlet classes must implement the JspPage interface. It extends the Servlet interface. It provides two life cycle methods.



Methods of JspPage interface

public void jspInit (): It is invoked only once during the life cycle of the JSP when JSP page is requested firstly. It is used to perform initialization. It is same as the init () method of Servlet interface.

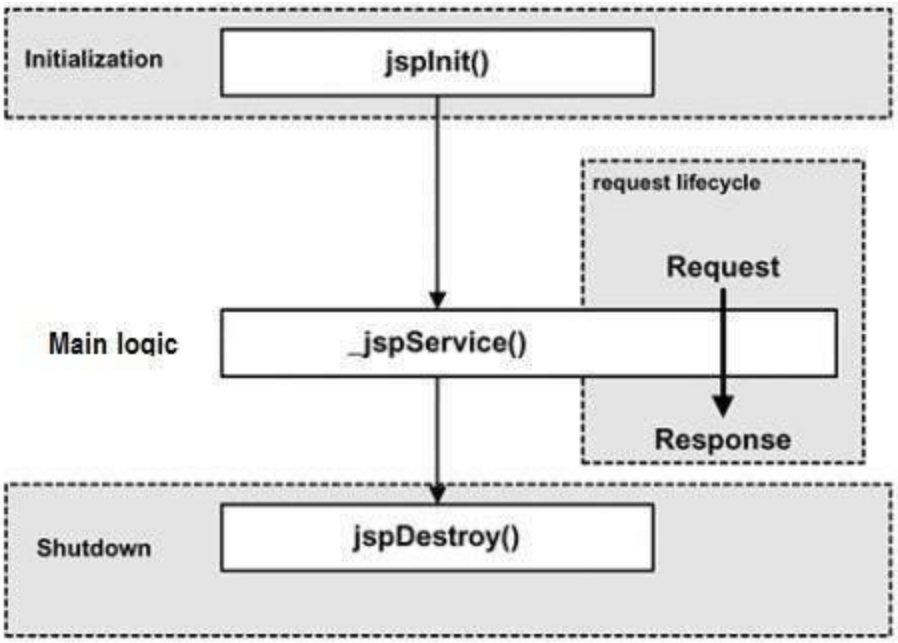
public void jspDestroy (): It is invoked only once during the life cycle of the JSP before the JSP page is destroyed. It can be used to perform some clean up operation.

18.3 JSP Lifecycle

The following are the paths followed by a JSP:

- Compilation
- Initialization
- Execution
- Cleanup

The four major phases of a JSP life cycle are very similar to the Servlet Life Cycle. The four phases have been described below –



JSP Compilation

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page.

The compilation process involves three steps −

- Parsing the JSP.
- Turning the JSP into a servlet.
- Compiling the servlet.

JSP Initialization

When a container loads a JSP it invokes the jspInit () method before servicing any requests. If you need to perform JSP-specific initialization, override the jspInit () method:

```
public void jspInit() {
   // Initialization code...
}
```

Typically, initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

JSP Execution

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.

Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the _jspService() method in the JSP.

The _jspService() method takes an HttpServletRequest and an HttpServletResponse as its parameters as follows:

```
void _jspService(HttpServletRequest request, HttpServletResponse
response) {
   // Service handling code...
}
```

The _jspService() method of a JSP is invoked on request basis. This is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods, i.e., GET, POST, DELETE, etc.

JSP Cleanup

The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.

The jspDestroy() method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.

The jspDestroy() method has the following form:

```
public void jspDestroy() {
   // Your cleanup code goes here.
}
```

18.4 Session Handling

JSP makes use of the servlet provided HttpSession Interface. This interface provides a way to identify a user across.

- a one-page request or
- visit to a website or
- store information about that user

By default, JSPs have session tracking enabled and a new HttpSession object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows:

```
<%@ page session = "false" %>
```

The JSP engine exposes the HttpSession object to the JSP author through the implicit session object. Since session object is already provided to the JSP programmer, the programmer can immediately begin storing and retrieving data from the object without any initialization or getSession().

Here is a summary of important methods available through the session object:

No.	Method & Description
1	**public Object getAttribute(String name)** This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	**public Enumeration getAttributeNames()** This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3	**public long getCreationTime()** This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4	**public String getId()** This method returns a string containing the unique identifier assigned to this session.
5	**public long getLastAccessedTime()** This method returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
6	**public int getMaxInactiveInterval ()**

	This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
7	**public void invalidate ()** This method invalidates this session and unbinds any objects bound to it.
8	**public boolean isNew ()** This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
9	**public void removeAttribute (String name)** This method removes the object bound with the specified name from this session.
10	**public void setAttribute (String name, Object value)** This method binds an object to this session, using the name specified.
11	**public void setMaxInactiveInterval (int interval)** This method specifies the time, in seconds, between client requests before the servlet con tainer will invalidate this session.

Session Tracking Example

This example describes how to use the **HttpSession** object to find out the creation time and the last-accessed time for a session. We would associate a new session with the request if one does not already exist.

```
<%@ page import = "java.io.*,java.util.*" %>
<%
// Get session creation time.
Date createTime = new Date(session.getCreationTime());

// Get last access time of this Webpage.
Date lastAccessTime = new Date(session.getLastAccessedTime());

String title = "Welcome Back to my website";
Integer visitCount = new Integer(0);
String visitCountKey = new String("visitCount");
String userIDKey = new String("userID");
String userID = new String("ABCD");

// Check if this is new comer on your Webpage.
if (session.isNew()) {
    title = "Welcome to my website";
    session.setAttribute(userIDKey, userID);
    session.setAttribute(visitCountKey, visitCount);
}

visitCount = (Integer) session.getAttribute(visitCountKey);
visitCount = visitCount + 1;
userID = (String) session.getAttribute(userIDKey);
session.setAttribute(visitCountKey, visitCount);
%>
```

```
<html>
  <head>
    <title>Session Tracking</title>
  </head>

  <body>
    <center>
      <h1>Session Tracking</h1>
    </center>

    <table border="1" align="center">
      <tr bgcolor="#949494">
        <th>Session info</th>
        <th>Value</th>
      </tr>
      <tr>
        <td>id</td>
        <td><% out.print( session.getId()); %></td>
      </tr>
      <tr>
        <td>Creation Time</td>
        <td><% out.print(createTime); %></td>
      </tr>
      <tr>
        <td>Time of Last Access</td>
        <td><% out.print(lastAccessTime); %></td>
      </tr>
      <tr>
        <td>User ID</td>
        <td><% out.print(userID); %></td>
      </tr>
      <tr>
        <td>Number of visits</td>
        <td><% out.print(visitCount); %></td>
      </tr>
    </table>
  </body>
</html>
```

18.5 Cookie Handling

Following table lists out the useful methods associated with the Cookie object which you can use while manipulating cookies in JSP:

No.	Method & Description
1	**public void setDomain(String pattern)** This method sets the domain to which the cookie applies; for example, jomdemy.com.com.
2	**public String getDomain ()** This method gets the domain to which the cookie applies; for example, jomdemy.com.
3	**public void setMaxAge (int expiry)** This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
4	**public int getMaxAge ()** This method returns the maximum age of the cookie, specified in seconds, by default, **-1** indicating the cookie will persist until the browser shutdown.
5	**public String getName ()** This method returns the name of the cookie. The name cannot be changed after the creation.
6	**public void setValue (String newValue)** This method sets the value associated with the cookie.
7	**public String getValue ()** This method gets the value associated with the cookie.
8	**public void setPath (String uri)** This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.

9	**public String getPath ()** This method gets the path to which this cookie applies.
10	**public void setSecure (boolean flag)** This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e., SSL) connections.
11	**public void setComment (String purpose)** This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.
12	**public String getComment ()** This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.

Setting Cookies with JSP

Setting cookies with JSP involves three steps:

Step 1: Creating a Cookie object

You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

```
Cookie cookie = new Cookie ("key”,” value");
```

Keep in mind, neither the name nor the value should contain white space or any of the following characters:

[ ] ( ) = , " / ? @ : ;

Step 2: Setting the maximum age

You use setMaxAge to specify how long (in seconds) the cookie should be valid. The following code will set up a cookie for 24 hours.

```
cookie. setMaxAge (60*60*24);
```

Step 3: Sending the Cookie into the HTTP response headers

You use response.addCookie to add cookies in the HTTP response header as follows

```
response.addCookie(cookie);
```

Example, Setting Cookies

```
<%
   // Create cookies for first and last names.
   Cookie firstName = new Cookie("first_name", request.getParameter("first_name"));
   Cookie lastName = new Cookie("last_name", request.getParameter("last_name"));

   // Set expiry date after 24 Hrs for both the cookies.
   firstName.setMaxAge(60*60*24);
   lastName.setMaxAge(60*60*24);

   // Add both the cookies in the response header.
   response.addCookie( firstName );
   response.addCookie( lastName );
%>

<html>
   <head>
      <title>Setting Cookies</title>
   </head>

   <body>
      <center>
         <h1>Setting Cookies</h1>
      </center>
      <ul>
         <li><p><b>First Name:</b>
            <%= request.getParameter("first_name")%>
         </p></li>
         <li><p><b>Last  Name:</b>
            <%= request.getParameter("last_name")%>
         </p></li>
      </ul>

   </body>
</html>
```

Reading Cookies

```
<html>
   <head>
      <title>Reading Cookies</title>
   </head>

   <body>
      <center>
         <h1>Reading Cookies</h1>
      </center>
      <%
         Cookie cookie = null;
         Cookie[] cookies = null;

         // Get an array of Cookies associated with the this domain
         cookies = request.getCookies();

         if( cookies != null ) {
            out.println("<h2> Found Cookies Name and Value</h2>");

            for (int i = 0; i < cookies.length; i++) {
               cookie = cookies[i];
               out.print("Name : " + cookie.getName( ) + ",  ");
               out.print("Value: " + cookie.getValue( )+" <br/>");
            }
         } else {
            out.println("<h2>No cookies founds</h2>");
         }
      %>
   </body>

</html>
```

18.6 Request

The **JSP request** is an implicit object of type HttpServletRequest i.e. created for each jsp request by the web container. It can be used to get request information such as parameter, header information, remote address, server name, server port, content type, character encoding etc.

It can also be used to set, get and remove attributes from the jsp request scope.

Let's see the simple example of request implicit object where we are printing the name of the user with welcome message.

Example of JSP request implicit object

index.html

```
<form action="welcome.jsp">
    <input type="text" name="uname">
    <input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
String name=request.getParameter("uname");
out.print("welcome "+name);
%>
```

Output

18.7 Response

In JSP, response is an implicit object of type HttpServletResponse. The instance of HttpServletResponse is created by the web container for each jsp request.

It can be used to add or manipulate response such as redirect response to another resource, send error etc.

Let's see the example of response implicit object where we are redirecting the response to the Google.

Example of response implicit object

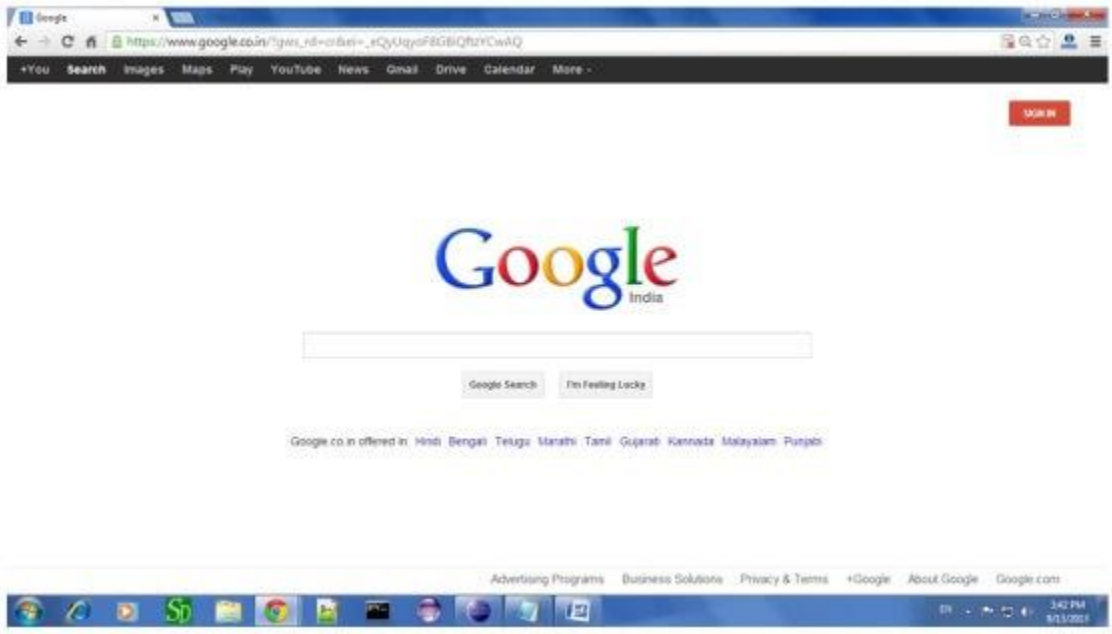
index.html

```
<form action="welcome.jsp">
    <input type="text" name="uname">
    <input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
response.sendRedirect("http://www.google.com");
%>
```

Output



18.8 Scriptlets

In JSP, java code can be written inside the jsp page using the scriptlet tag. Let's see what the scripting elements are first.

JSP Scripting elements

The scripting elements provides the ability to insert java code inside the jsp. There are three types of scripting elements:

- scriptlet tag
- expression tag
- declaration tag
- JSP scriptlet tag

A scriptlet tag is used to execute java source code in JSP. Syntax is as follows:

```
<% java source code %>
```

Example of JSP scriptlet tag

In this example, we are displaying a welcome message.

```
<html>
<body>
<% out.print("welcome to jsp"); %>
</body>
</html>
```

18.9 Expressions

The code placed within JSP expression tag is written to the output stream of the response. So, you need not write out.print() to write data. It is mainly used to print the values of variable or method.

Syntax of JSP expression tag

```
<%= statement %>
```

Example of JSP expression tag

In this example of jsp expression tag, we are simply displaying a welcome message.

```
<html>
<body>
<%= "welcome to jsp" %>
</body>
</html>
```

18.10 Declarations

The JSP declaration tag is used to declare fields and methods. The code written inside the jsp declaration tag is placed outside the service() method of auto generated servlet. So it doesn't get memory at each request.

Syntax of JSP declaration tag

The syntax of the declaration tag is as follows:

```
<%! field or method declaration %>
```

Example of JSP declaration tag that declares field

In this example of JSP declaration tag, we are declaring the field and printing the value of the declared field using the jsp expression tag.

index.jsp

```
<html>
<body>
<%! int data=50; %>
<%= "Value of the variable is:"+data %>
</body>
</html>
```

18.11 What Are Directives In JSP?

The jsp directives are messages that tells the web container how to translate a JSP page into the corresponding servlet.

There are three types of directives:

- page directive
- include directive
- taglib directive

Syntax of JSP Directive

```
<%@ directive attribute=”value” %>
```

JSP page directive

The page directive defines attributes that apply to an entire JSP page.

Syntax of JSP page directive

```
<%@ page attribute="value" %>
```

Attributes of JSP page directive

- import
- contentType
- extends
- info
- buffer
- language
- isELIgnored
- isThreadSafe
- autoFlush
- session
- pageEncoding
- errorPage
- isErrorPage

1) import

The import attribute is used to import class, interface or all the members of a package. It is similar to import keyword in java class or interface.

Example of import attribute

```
<html>
<body>

<%@ page import="java.util.Date" %>
Today is: <%= new Date() %>

</body>
</html>
```

2) contentType

The contentType attribute defines the MIME(Multipurpose Internet Mail Extension) type of the HTTP response.The default value is "text/HTML; charset=ISO-8859-1".

Example of contentType attribute

```
<html>
<body>

<%@ page contentType=application/msword %>
Today is: <%= new java.util.Date() %>

</body>
</html>
```

3) extends

The extends attribute defines the parent class that will be inherited by the generated servlet.It is rarely used.

4) info

This attribute simply sets the information of the JSP page which is retrieved later by using getServletInfo() method of Servlet interface.

Example of info attribute

```
<html>
<body>

<%@ page info="composed by Azman" %>
Today is: <%= new java.util.Date() %>

</body>
</html>
```

5) buffer

The buffer attribute sets the buffer size in kilobytes to handle output generated by the JSP page.The default size of the buffer is 8Kb.

Example of buffer attribute

```
<html>
<body>

<%@ page buffer="16kb" %>
Today is: <%= new java.util.Date() %>

</body>
</html>
```

6) language

The language attribute specifies the scripting language used in the JSP page. The default value is "java".

7) isELIgnored

We can ignore the Expression Language (EL) in jsp by the isELIgnored attribute. By default its value is false i.e. Expression Language is enabled by default. We see Expression Language later.

```
<%@ page isELIgnored="true" %>//Now EL will be ignored
```

8) isThreadSafe

Servlet and JSP both are multithreaded.If you want to control this behaviour of JSP page, you can use isThreadSafe attribute of page directive.The value of isThreadSafe value is true.If you make it false, the web container will serialize the multiple requests, i.e. it will wait until the JSP finishes responding to a request before passing another request to it.If you make the value of isTh readSafe attribute like:

```
<%@ page isThreadSafe="false" %>
```

The web container in such a case, will generate the servlet as:

```
public class SimplePage_jsp extends HttpJspBase
 implements SingleThreadModel{
      .......
 }
```

9) errorPage

The errorPage attribute is used to define the error page, if exception occurs in the current page, it will be redirected to the error page.

```
<html>
<body>

<%@ page errorPage="myerrorpage.jsp" %>

 <%= 100/0 %>

</body>
</html>
```

10) isErrorPage

The isErrorPage attribute is used to declare that the current page is the error page.

Note: The exception object can only be used in the error page.

Example of isErrorPage attribute

```
<html>
<body>

<%@ page isErrorPage="true" %>

 Sorry an exception occured!<br/>
The exception is: <%= exception %>

</body>
</html>
```

18.12 Page Directive

The **page** directive is used to provide instructions to the container that pertain to the current JSP page. You may code the page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.

Following is the basic syntax of page directive –

```
<%@ page attribute = "value" %>
```

You can write the XML equivalent of the above syntax as follows –

```
<jsp:directive.page attribute = "value" />
```

Attributes

Following table lists out the attributes associated with the page directive –

No.	Attribute & Purpose
1	**buffer** Specifies a buffering model for the output stream.
2	**autoFlush** Controls the behavior of the servlet output buffer.
3	**contentType** Defines the character encoding scheme.
4	**errorPage** Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
5	**isErrorPage** Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.
6	**extends** Specifies a superclass that the generated servlet must extend.
7	**import** Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
8	**info** Defines a string that can be accessed with the servlet's **getServletInfo()** method.
9	**isThreadSafe** Defines the threading model for the generated servlet.

10	**language** Defines the programming language used in the JSP page.
11	**session** Specifies whether or not the JSP page participates in HTTP sessions.
12	**isELIgnored** Specifies whether or not the EL expression within the JSP page will be ignored.
13	**isScriptingEnabled** Determines if the scripting elements are allowed for use.

18.13 Include Directives

The include directive is used to include the contents of any resource it may be jsp file, html file or text file. The include directive includes the original content of the included resource at page translation time (the jsp page is translated only once so it will be better to include static resource).

Advantage of Include directive

- Code Reusability

Syntax of include directive

```
<%@ include file="resourceName" %>
```

Example of include directive

In this example, we are including the content of the header.html file. To run this example you must create an header.html file.

```
<html>
<body>

<%@ include file="header.html" %>

Today is: <%= java.util.Calendar.getInstance().getTime() %>

</body>
</html>
```

18.14 Taglib Directive

The JSP taglib directive is used to define a tag library that defines many tags. We use the TLD (Tag Library Descriptor) file to define the tags. In the custom tag section we will use this tag so it will be better to learn it in custom tag.

Syntax JSP Taglib directive

```
<%@ taglib uri="uriofthetaglibrary" prefix="prefixoftaglibrary" %>
```

Example of JSP Taglib directive

In this example, we are using our tag named currentDate. To use this tag we must specify the taglib directive so the container may get information about the tag.

```
<html>
<body>

<%@ taglib uri="http://www.javatpoint.com/tags" prefix="mytag" %>

<mytag:currentDate/>

</body>
</html>
```

Web Services

In this module, you will be able to learn java web services and its specifications such as JAX-WS and JAX-RS. There are two ways to write the code for JAX-WS by RPC style and Document style. Like JAX-WS, JAX-RS can be written by Jersey and RESTeasy.

19.1 JAX-WS

There are two ways to develop JAX-WS example.

- RPC style
- Document style

Difference between RPC and Document web services

There are many differences between RPC and Document web services. The important differences between RPC and Document are given below:

RPC Style

- RPC style web services use method name and parameters to generate XML structure.
- The generated WSDL is difficult to be validated against schema.
- In RPC style, SOAP message is sent as many elements.
- RPC style message is tightly coupled.
- In RPC style, SOAP message keeps the operation name.
- In RPC style, parameters are sent as discrete values.

Let's see the RPC style generated WSDL file.

WSDL file:

In WSDL file, it doesn't specify the types details.

For message part, it defines name and type attributes.

```
<message name="getHelloWorldAsString">
    <part name="arg0" type="xsd:string" />
</message>

<message name="getHelloWorldAsStringResponse">
    <part name="return" type="xsd:string" />
</message>
```

For soap:body, it defines use and namespace attributes.

```
<binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc" />
    <operation name="getHelloWorldAsString">
        <soap:operation soapAction="" />
        <input>
            <soap:body use="literal" namespace="http://jomdemy.com/" />
        </input>
        <output>
            <soap:body use="literal" namespace="http://jomdemy.com/" />
        </output>
    </operation>
</binding>
```

Document Style

- Document style web services can be validated against predefined schema.
- In document style, SOAP message is sent as a single document.
- Document style message is loosely coupled.
- In Document style, SOAP message loses the operation name.
- In Document style, parameters are sent in XML format.

Let's see the Document style generated WSDL file.

```
<types>
    <xsd:schema>
        <xsd:import namespace="http://javatpoint.com/" schemaLocation="http://localhost:7779/ws/hello?xsd=1" />
    </xsd:schema>
</types>
```

For message part, it defines name and element attributes.

```
<message name="getHelloWorldAsString">
    <part name="parameters" element="tns:getHelloWorldAsString" />
</message>

<message name="getHelloWorldAsStringResponse">
    <part name="parameters" element="tns:getHelloWorldAsStringResponse" />
</message>
```

For soap:body, it defines use attribute only not namespace.

```
<binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
    <operation name="getHelloWorldAsString">
        <soap:operation soapAction="" />
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />
        </output>
    </operation>
</binding>
```

JAX-WS Example RPC Style

Creating JAX-WS example is a easy task because it requires no extra configuration settings.

JAX-WS API is inbuilt in JDK, so you don't need to load any extra jar file for it. Let's see a simple example of JAX-WS example in RPC style.

There are created 4 files for hello world JAX-WS example:

- HelloWorld.java
- HelloWorldImpl.java
- Publisher.java
- HelloWorldClient.java

The first 3 files are created for server side and 1 application for client side.

```java
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.xml.ws.Endpoint;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

//Service Endpoint Interface
@WebService
@SOAPBinding(style = Style.RPC)
public interface HelloWorld {
    @WebMethod
    String getHelloWorldAsString(String name);
}

// Service Implementation
@WebService(endpointInterface = "com.jomdemy.HelloWorld")
class HelloWorldImpl implements HelloWorld {
    @Override
    public String getHelloWorldAsString(String name) {
        return "Hello World JAX-WS " + name;
    }
}

class HelloWorldPublisher {
    Run | Debug
    public static void main(String[] args) {
        Endpoint.publish(address: "http://localhost:7779/ws/hello", new HelloWorldImpl());
    }
}

class HelloWorldClient {
    Run | Debug
    public static void main(String[] args) throws Exception {
        URL url = new URL(spec: "http://localhost:7779/ws/hello?wsdl");

        // 1st argument service URI, refer to wsdl document above
        // 2nd argument is service name, refer to wsdl document above
        QName qname = new QName(namespaceURI: "http://jomdemy.com/", localPart: "HelloWorldImplService");
        Service service = Service.create(url, qname);
        HelloWorld hello = service.getPort(serviceEndpointInterface: HelloWorld.class);
        System.out.println(hello.getHelloWorldAsString(name: "javatpoint rpc"));
    }
}
```

How to view generated WSDL

After running the publisher code, you can see the generated WSDL file by visiting the URL:

http://localhost:7779/ws/hello?wsdl

19.2 JAX-RS

This JAX-RS topic is designed for beginners and professionals. There are two main implementation of JAX-RS API.

- Jersey
- RESTEasy

19.3 Jersey

We can create JAX-RS example by jersey implementation. To do so, you need to load jersey jar files or use maven framework. In this example, we are using jersey jar files for using jersey example for JAX-RS.

There are created 4 files for hello world JAX-RS example:

- Hello.java
- web.xml
- index.html
- HelloWorldClient.java

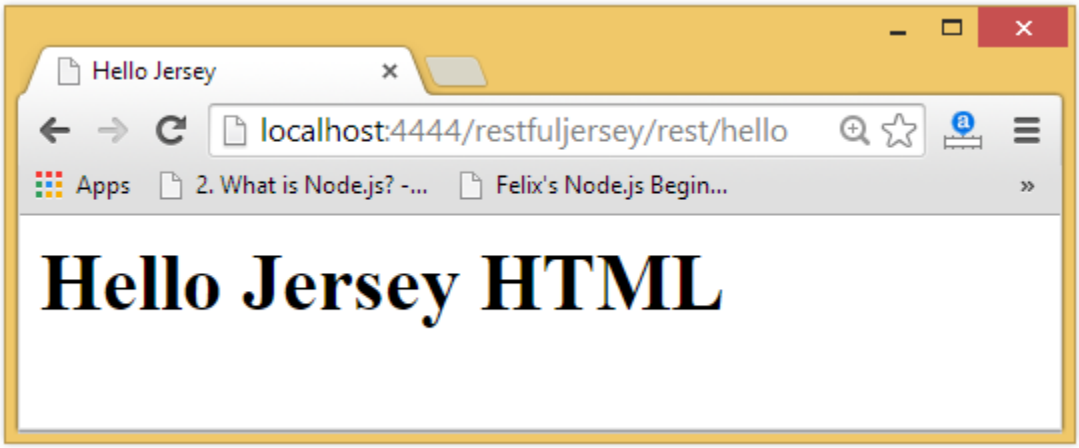
The first 3 files are created for server side and 1 application for client side.

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/hello")
public class Hello {
    // This method is called if HTML and XML is not requested
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayPlainTextHello() {
        return "Hello Jersey Plain";
    }

    // This method is called if XML is requested
    @GET
    @Produces(MediaType.TEXT_XML)
    public String sayXMLHello() {
        return "<?xml version=\"1.0\"?>" + "<hello> Hello Jersey" + "</hello>";
    }

    // This method is called if HTML is requested
    @GET
    @Produces(MediaType.TEXT_HTML)
    public String sayHtmlHello() {
        return "<html> " + "<title>" + "Hello Jersey" + "</title>"
                + "<body><h1>" + "Hello Jersey HTML" + "</h1></body>" + "</html> ";
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
    <servlet>
        <servlet-name>Jersey REST Service</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>com.javatpoint.rest</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Jersey REST Service</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Now run this application on server. Here we are using Tomcat server on port 4444. The project name is restfuljersey.

After running the project, you will see the following output:



JAX-RS Client Code

The ClientTest.java file is created inside the server application. But you can run client code by other application also by having service interface and jersey jar file.

File: ClientTest.java

```
import java.net.URI;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.UriBuilder;
import org.glassfish.jersey.client.ClientConfig;

public class ClientTest {
    Run | Debug
    public static void main(String[] args) {
        ClientConfig config = new ClientConfig();
        Client client = ClientBuilder.newClient();
        WebTarget target = client.target(getBaseURI());
        // Now printing the server code of different media type
        System.out.println(target.path(path: "rest").path(path: "hello").request().accept(MediaType.TEXT_PLAIN).get(responseType: String.class));
        System.out.println(target.path(path: "rest").path(path: "hello").request().accept(MediaType.TEXT_XML).get(responseType: String.class));
        System.out.println(target.path(path: "rest").path(path: "hello").request().accept(MediaType.TEXT_HTML).get(responseType: String.class));
    }

    private static URI getBaseURI() {
        // here server is running on 4444 port number and project name is restfuljersey
        return UriBuilder.fromUri("http://localhost:4444/restfuljersey").build();
    }
}
```

Output

Hello Jersey Plain

<?xml version="1.0"?><hello> Hello Jersey</hello>

<html> <title>Hello Jersey</title><body><h1>Hello Jersey HTML</h1></body></html>

19.4 Example

This will be hands-on example together with your instructor

Java IO

- 20.1 Introduction
- 20.2 Bytes Stream
- 20.3 Character Streams

20.1 Introduction

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams:

- InPutStream – The InputStream is used to read data from a source.
- OutPutStream – The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this module covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one:

20.2 Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream. Following is an example which makes use of these two classes to copy an input file into an output file:

Example

```java
import java.io.*;

public class CopyFile {

    Run | Debug
    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in  = new FileInputStream(name: "data/input.txt");
            out = new FileOutputStream(name: "data/output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }

            if (out != null) {
                out.close();
            }
        }
    }
}
```

20.3 Character Stream

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file

```
class CopyFile2 {

   Run | Debug
   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader(fileName: "data/input.txt");
         out = new FileWriter(fileName: "data/output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

20.4 Standard Stream

Java provides the following three standard streams:

- **Standard Input** – This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.

- **Standard Output** – This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as System.out.

- **Standard Error** – This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.

Following is a simple program, which creates InputStreamReader to read standard input stream until the user types a "q":

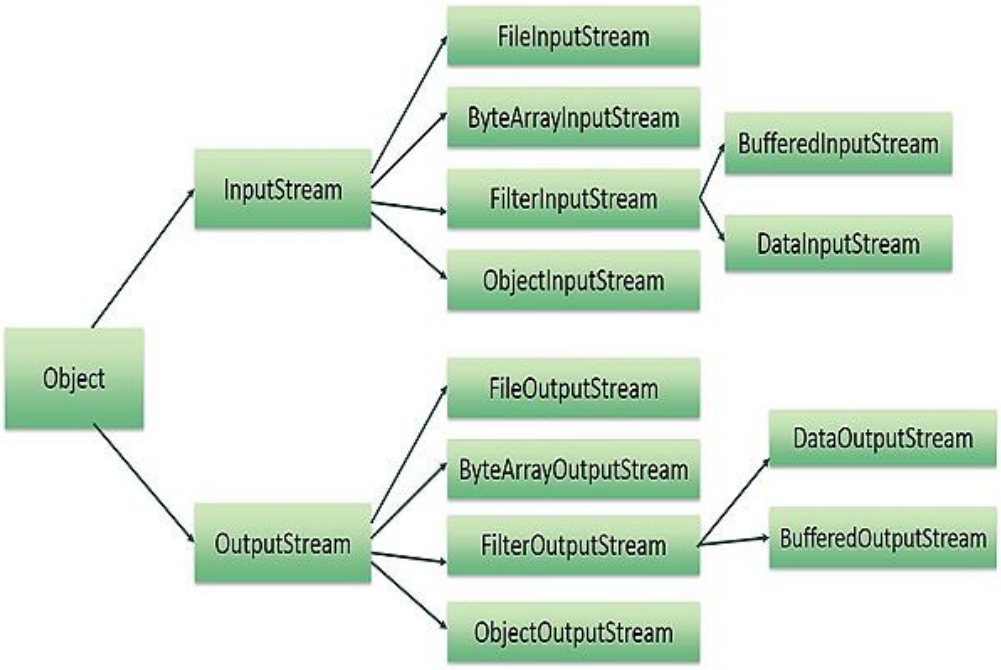
Example

```
import java.io.*;
public class ReadConsole {

   Run | Debug
   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println(x: "Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

20.5 Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



20.6 File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java

A directory is a File which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories

There are two useful File utility methods, which can be used to create directories –

- The mkdir( ) method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.

- The mkdirs() method creates both a directory and all the parents of the directory.

Following example creates "/data/user/java/bin" directory:

Example

```
import java.io.File;
public class CreateDir {

    Run | Debug
    public static void main(String args[]) {
        String dirname = "data/tmp/user/java/bin";
        File d = new File(dirname);
        // Create directory now.
        d.mkdirs();
    }
}
```

Listing Directories

You can use list( ) method provided by File object to list down all the files and directories available in a directory as follows:

Example

```
import java.io.File;
public class ReadDir {

   Run | Debug
   public static void main(String[] args) {
      File file = null;
      String[] paths;

      try {
         // create new file object
         file = new File(pathname: "data");

         // array of files and directory
         paths = file.list();

         // for each name in the path array
         for(String path:paths) {
            // prints filename and directory name
            System.out.println(path);
         }
      } catch (Exception e) {
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

Java Threads

Threads allows a program to operate more efficiently by doing multiple things at the same time.

Threads can be used to perform complicated tasks in the background without interrupting the main program.

Creating a Thread

There are two ways to create a thread.

It can be created by extending the `Thread` class and overriding its `run()` method:

Extend Syntax

```
public class Main extends Thread {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Another way to create a thread is to implement the `Runnable` interface:

Implement Syntax

```
public class Main implements Runnable {
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Running Threads

If the class extends the `Thread` class, the thread can be run by creating an instance of the class and call its `start()` method:

Extend Example

```
public class Main extends Thread {

  public static void main(String[] args) {

    Main thread = new Main();

    thread.start();

    System.out.println("This code is outside of the thread");

  }

  public void run() {

    System.out.println("This code is running in a thread");

  }

}
```

If the class implements the `Runnable` interface, the thread can be run by passing an instance of the class to a `Thread` object's constructor and then calling the thread's `start()` method:

Implement Example

```
public class Main implements Runnable {

  public static void main(String[] args) {

    Main obj = new Main();

    Thread thread = new Thread(obj);

    thread.start();

    System.out.println("This code is outside of the thread");

  }
```

```
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Differences between "extending" and "implementing" Threads

The major difference is that when a class extends the Thread class, you cannot extend any other class, but by implementing the Runnable interface, it is possible to extend from another class as well, like: class `MyClass extends OtherClass implements Runnable`.

Concurrency Problems

Because threads run at the same time as other parts of the program, there is no way to know in which order the code will run. When the threads and main program are reading and writing the same variables, the values are unpredictable. The problems that result from this are called concurrency problems.

Example

A code example where the value of the variable **amount** is unpredictable:

```
public class Main extends Thread {
  public static int amount = 0;

  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println(amount);
    amount++;
    System.out.println(amount);
```

```
  }


  public void run() {
    amount++;
  }
}
```

To avoid concurrency problems, it is best to share as few attributes between threads as possible. If attributes need to be shared, one possible solution is to use the `isAlive()` method of the thread to check whether the thread has finished running before using any attributes that the thread can change.

Example

Use `isAlive()` to prevent concurrency problems:

```
public class Main extends Thread {
  public static int amount = 0;


  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    // Wait for the thread to finish
    while(thread.isAlive()) {
    System.out.println("Waiting...");
  }
  // Update amount and print its value
  System.out.println("Main: " + amount);
  amount++;
  System.out.println("Main: " + amount);
```

```
  }
  public void run() {
    amount++;
  }
}
```