

# Object-Oriented Programming

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- History of object-oriented programming.
- Principles of object-oriented programming.
- Encapsulation and how it works in practice.
- Inheritance and how it works in practice.
- Polymorphism and how it works in practice.
- Abstraction and how it works in practice.
- Abstract class and how to use it in your program.
- Interface and how it is useful in designing inheritance.
- Future of object-oriented programming.
- Overloading and its use.
- Overriding and its use.

## 12.1 | Introduction

Programming languages allow computers to offer varied functionality. They enable us to create specialized applications that are capable of providing advanced functionality. There are different programming schemes that are used by different programming language platforms. This chapter focuses on the principles of object-oriented programming, or OOP in short.

This chapter is divided into relevant sections. We will start by discussing basic OOP principles, which define the use of objects in programming languages such as Java. We will then discuss the use of classes and objects for implementing the same functionality, as produced by more direct languages such as C. We will discuss various subtopics that are essential for building a key understanding of the OOP world.

We will first discuss the basic principles of OOP languages and then study the other important themes that you must explore to learn more about the OOP world. Let us look at the history of programming principles, which shows how it was gradually possible to reach the OOP design of programming paradigm.

### 12.1.1 History of Object-Oriented Programming

The concepts of using objects and providing an orientation to computer code were developed in the early 1960s, especially by researchers at MIT. Programmers referred to code elements as objects, which have a set of properties. Simula is the first language that presented the concepts of classes and objects, which are important for defining a language as one that employs OOP principles.

OOP languages improved data security and allowed programmers to produce data encapsulation in the form of creating private and public variables. Such compilers became popular and were employed to create programs for mainframe computers. The early OOP tools were especially efficient when used for carrying out complex tasks.

OOP principles especially got in popular use as they were perfect for creating programs that could follow human language-form instructions. The concepts of inheritance and encapsulation really caught on with various programmer communities, which understood that there were several benefits of switching from function-based programming especially for complex tasks.

The early languages started with smart ways to describe objects and create situations where it is possible to hide the implementation of the code to other programmers and program users. The programming environments gradually became available to smaller working situations.

There are two approaches that became widely used, with other approaches being dropped. Functional programming and OOP became the paradigms that were employed in all popular languages. There are many differences between functional

programming and OOP. One of the major differences is that functional programming follows stateless programming model and OOP follows stateful programming model.

C++ is an object-oriented language that was prepared from C, which is a fundamentally functional programming language. Software construction still followed instances where functional languages formed the basis of development while allowing programmers to use tools and libraries that implemented objectivity in their programming use.

OOP concepts gradually became popular as it was important to improve code security and provide control over data interfaces and class implementations. The OOP concepts gave rise to design patterns that are now commonly employed to resolve software problems in the modern development environment.

The use of behaviors and inheritance are the primary factors to incorporate an object-oriented design. This is a situation where it is possible to employ polymorphism and ensure the use of mutable objects. There are various methods that are still under the basic foundation of object-oriented principles. This may include abstraction, prototyping, and the use of singleton structures.

OOP languages have already surpassed the use of functional languages in modern use. These days, they are competing with the use of relational databases, as this ensures that it is possible to resolve all issues. According to computer scholars, the OOP paradigm is perfect when developers must create software systems that resemble human elements. It is perfect for taking a natural approach towards resolving problems by creating objects, which achieve the objectives required to resolve each scenario.

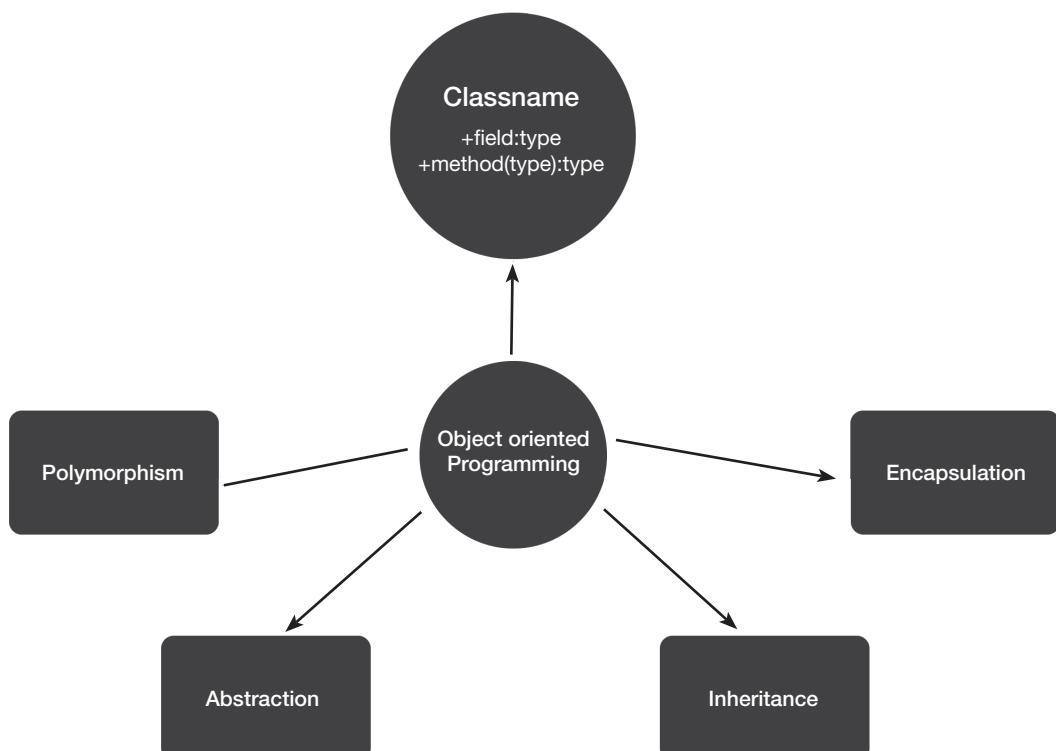
### QUICK CHALLENGE

Create a comparison chart to differentiate between procedural programming and object-oriented programming.



## 12.2 | Object-Oriented Programming Principles

OOP languages use the concept of creating every functionality with the use of distinct objects. This is different from the principle of creating functions that hold independent value. OOP principles call for creating data objects that provide a higher level of functionality and make it easier for the program developer to prepare a project according to the specific requirements of the client. Figure 12.1 shows the object-oriented principles.



**Figure 12.1** Object-oriented programming principles.

Regardless of any OOP language, there are four principles that define this technique.

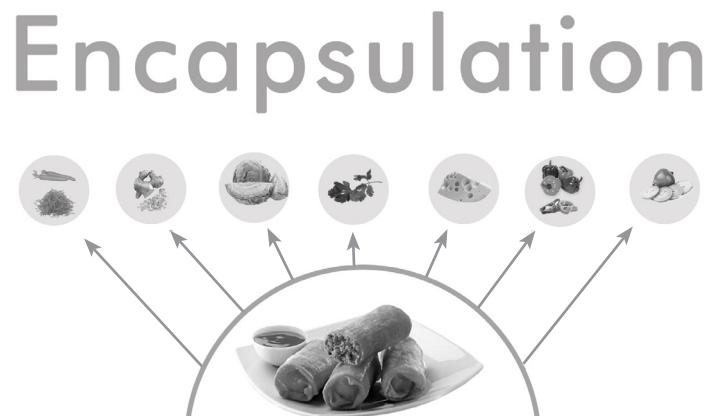
- 1. Encapsulation:** This is a method that separates data implementation from the user.
- 2. Abstraction:** It describes the use of simple class objects that may provide the most complex of functions. This is a key principle of these languages and is often combined with encapsulation for an easier understanding and practical application.
- 3. Inheritance:** It allows the creation of data hierarchy structures, which ensures that data objects can form related trees and branches. It creates the system of classes that are built within other classes to follow a systematic relationship for defining instances, variables, and implementing the functionality from all the upper level classes. In other words, every class that has super class gets access to variables and methods from all the super classes at every level.
- 4. Polymorphism:** It is a more difficult concept to understand. In simple terms, it means that a single object may take on different forms, according to the defining principles of its use in the programming language.

Now, we will discuss these important concepts in greater detail. This will help you build a better understanding of the OOP concepts, and allow you to figure out how to go about using OOP languages for programming and application development.

### 12.2.1 Encapsulation

The first concept is the encapsulation of the data. Since data is arranged in the form of defined objects, they hold the properties of being distinct in their structure and is fully self-contained. The inner working of an object is defined by its state (attributes) and remains invisible to other parts of the code. The objects can show their behavior but maintain a strong boundary that separates them from other objects that are present in the programming environment.

Encapsulation works on the principle of hiding. The inner structure of all data objects is distinct. It contains all the elements, which are required to process it as a standalone part of the programming code. This objective is achieved by implementing boundaries that protect objects using specific tools. Access modifiers are used in a language such as Java, which allows you to hold full control of the attributes that define a data object. Figure 12.2 shows an example of encapsulation where all the ingredients are hidden inside the spring role. This can help you to visualize how encapsulation works.



**Figure 12.2** Encapsulation example.

There are some languages that create very strong encapsulation boundaries. Then there are languages like Java, which provide better control over the object structure by allowing programmers to set up different object property specifiers. However, encapsulation is still a primary feature in Java too.

This important principle allows us to separately keep data and the code sections that call for it. It allows programmers to change the original code, while never affecting the database objects that hold the legacy data, since they are always called but remain inaccessible for change due to a specific access structure.

Here is an example that shows how we implement the principle of encapsulation in Java by producing functionality, which remains locked with a boundary that separates the outside world from the class behaviors.

```

package java11.fundamentals.chapter12;
class DemoEncap {
    private int ssnValue;
    private int employeeAge;
    private String employeeName;
    // We will employ get and set methods to use the class objects
    public int getEmployeeSSN() {
        return ssnValue;
    }
    public String getEmployeeName() {
        return employeeName;
    }
    public int getEmployeeAge() {
        return employeeAge;
    }
    public void setEmployeeAge(int newValue) {
        employeeAge = newValue;
    }
    public void setEmployeeName(String newValue) {
        employeeName = newValue;
    }
    public void setEmployeeSSN(int newValue) {
        ssnValue = newValue;
    }
}
public class TestEncapsulation {
    public static void main(String args[]) {
        DemoEncap obj = new DemoEncap();
        obj.setEmployeeName("Mark");
        obj.setEmployeeAge(30);
        obj.setEmployeeSSN(12345);
        System.out.println("Employee Name is: " + obj.getEmployeeName());
        System.out.println("Employee SSN Code is: " + obj.getEmployeeSSN());
        System.out.println("Employee Age is: " + obj.getEmployeeAge());
    }
}

```

In this example, we have three private variables that are described during class initialization. They remain private and cannot be affected by the methods used in a typical program. However, it is possible to use set and get methods to define these variables in a particular class, where they can be mentioned and used in the main method of the program. This will produce instances that create new data objects and hold value in them, while the actual initialization and the coding behind the variables remain separated in the class definition.

This program will print *Mark*, *30*, and *12345* from the object getting values, without ever affecting the definition of the variables or making it available for the program to alter them.

Encapsulation is slightly different from abstraction in the manner that it defines the combination of all concepts into a single item with the ability to hide its internal data, which is not directly accessible to a program user. Encapsulation creates low coupling where various code elements do not have to depend entirely on each other. This is an excellent programming practice, which efficiently uses the available resources and is achieved through this ideal OOP principle.

Encapsulation is also excellent in terms of allowing the data and functionality to remain available for a user, while still hiding the way it is implemented. There is no information about the way objects are supposed to work, while still understanding the data that it demands, and the methods that it contains to get the job done.

**QUICK  
CHALLENGE**

Based on the above-mentioned example of `DemoEncap`, write a program on a real-life problem to demonstrate encapsulation.

### 12.2.1.1 Advantages of Encapsulation

There are several advantages of encapsulation and they are the reasons for introducing languages that employ OOP principles. Here, we share the details of some important benefits:

1. It provides the advantage of creating flexible code. This is possible because we can implement a variable field in any way we want throughout the program. We can only use the different methods that remain within the classes that we have implemented. The class can be maintained directly, while various implementations occur as we see fit.
2. We can ensure that fields can be only either read or written. This is possible by avoiding the getter and setter methods. We can create a variable as a private one and then only use the get method. This will ensure that there will never be a change in the value of a particular field. Similarly, we will only use the set method to lock the variable value in only a written situation.
3. A program user will never know how the code works in the background for any variable present in an encapsulated class structure. This means that they only have access to employing get and set methods, where they want to bring value or set value to the variables. There is no way of finding out how the actual value would be assigned or read from the original variable, which remains private in a language like Java.



What are the disadvantages of encapsulation?

### 12.2.2 Abstraction

Abstraction is a concept which is best defined and described in line with encapsulation. It suggests that it is possible to develop classes and objects that are defined according to their functionality, rather than their programming implementation. This leaves us in terms of creating models for all our requirements.

These models serve as structures and never represent the presence of an actual item. The primary characteristics that define an object completely distinguish it from the other objects. Abstraction is also produced by creating conceptual boundaries for these defining characters, which ensures that all objects are understood by the viewers according to their specific perspective.

Abstraction means that we can create programming tools that we can employ multiple times. It defines an object without ever presenting the object for its instance. This can be understood by creating classes that effectively deal with related items such as recording the names and addresses of employees by using a class, which is designed for handling personal information.

Abstraction in Java can be best defined by using abstract classes. These are classes that you specify by adding the word “abstract” before mentioning the name of the class. This will lock the class and tell the compiler that it cannot be instantiated ahead. It is simply defining a class as an incomplete one. However, the incompleteness can be present in any part of the class definition. This is again identified with the keyword “abstract” with the method that we want to leave undefined. Here is a simple example:

```
abstract class MusicInstruments {
    protected String nameofInstrument;
    abstract public void playInst();
}
```

On its own, this will be an abstract object that has a defined structure but actually points to particular code or functionality in a program. However, we use this abstract by creating an extension that produces a subclass:

```
abstract class NewInstrument extends MusicInstruments {
    protected int numberOfPieces;
}
```

Now, this is a creation of a subclass that defines some object of value which has an additional field, aside from the initial variable fields. We can now create a set of subclasses that can all add various values and can be used for an actual implementation in a

Java program. Java also has the capability of producing interfaces, which allow for the use of the same abstraction principle to carry out the intended functionality that remains hidden from the common users.



Can a class extend multiple abstract classes?

### 12.2.2.1 Use of Abstract Classes

As we have seen above, Abstract classes set up abstract methods, which cannot have an implementation. You can have an abstract class which does not have any abstract methods. These classes lie between interface structures and the common class structures. You may again be thinking about the use of such classes, especially in relation to OOP programming.

They are created to work as the parent shell for derived classes, which will then contain the objects that will be actually processed during subsequent codes and calls in the program. This process is further shown by the following example:

```
package java11.fundamentals.chapter12;
public abstract class Animals {
    public void PrintInfo() {
        System.out.println(GetSound());
    }
    protected abstract String GetSound();
}
```

This shows that we have one defined method in the abstract class, while there is another abstract method of `GetSound()`. Our choice of not instantiating this method means that it can hold different forms based on the creation of descending classes, which may use it differently especially with the use of code overriding.

#### QUICK CHALLENGE

Should you use interfaces over abstract classes? Give reasons for your answer.

### 12.2.2.2 Real-World Understanding

The principle of abstraction is best understood by understanding the principle of driving a car. Let us consider that it is a company car, where the company director is authorized to use it in any intended manner. He only needs to “call” the car driving function by contacting the relevant driver and does not require any specific details of the car. On the other hand, the driver is responsible for carrying out the functions required for performing the actual driving.

This means that there are two types of features. The first are the properties or the attributes. The director must know about them, such as the registration number of a car and the name of the driver tasked with driving it. The set of information may include other specifying details such as the color of the vehicle and its comfort level.

The second type of feature is the functions that the called car can perform. It can start from a particular office and take the company employees to a worksite. This may include driving, refueling the car, and other intended functions that are termed as methods. The controlling director, which works here as the “calling code” does not need to know about how they will be performed. He will simply order them to be executed to achieve the required functionality.

This example represents abstraction at its best, where we describe how OOP languages work for programmers, developers, and end users. OOP principles such as abstraction certainly makes it easier to simplify complex tasks and break them according to the required problem-solving steps.

### 12.2.2.3 Benefits of Abstraction

Abstraction offers several benefits and therefore, it forms the basis of all OOP languages. Here are some excellent advantages of using this specific principle:

1. It creates a barrier, which protects the implementation layer from the code users.

2. It also offers flexibility in terms of later changing the way implementation is carried out. This may be done to improve the coupling structure to loosen it up. A loose system is beneficial, as it allows all involved parties to make the best use of a working contract created through an application interface.
3. It makes it easier to perform debugging and find out which working layer is at fault for a particular problem.
4. It can be delivered through interfaces, allowing the easy correction of code use by an end user or the identification of wrong implementation scheme placed by the developer.
5. It also allows to set up for a divide and conquer policy for breaking a large program into smaller sections, which are easier to correct and implement by setting up interfaces that connect different parts of the complex program.



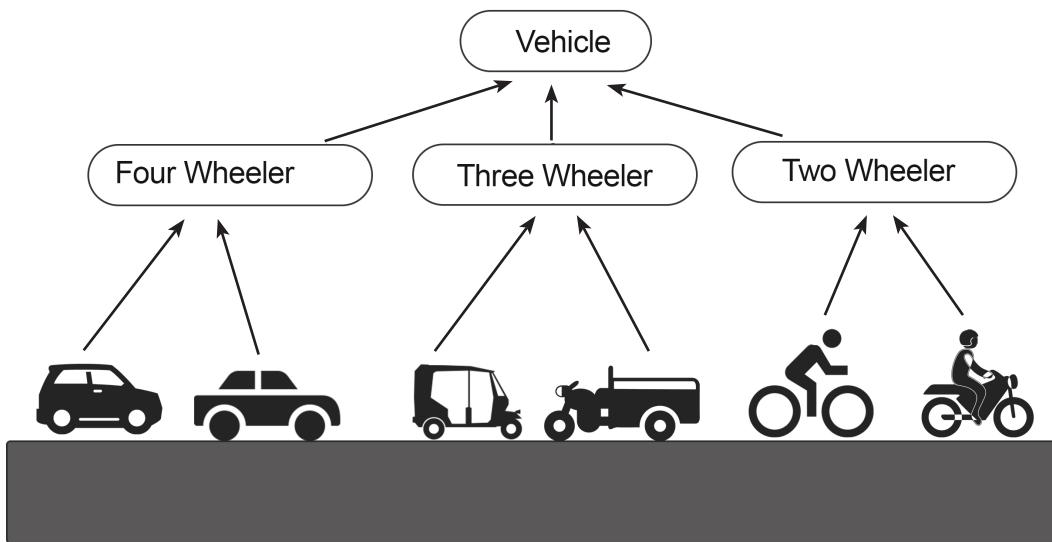
Does abstraction restrict the functionality of the implementing class?

### 12.2.3 Inheritance

This is a principle that is specifically designed to improve the performance of structured programming languages. However, these languages produce a lot of duplicate code in a long program, since many code structures are often required in various places. The OOP principle of inheritance solves this problem by creating hierachal sets of coding elements.

There are special classes that have the ability to copy the behaviors and attributes of their specialized functions. You can then only override the specific elements, which are required to be changed in different parts of the program. Your code becomes optimized, as each time the class object is called, a new specialization creates child classes and objects that only have the change in some of the elements.

The main class is termed as the parent class, which contains the source code. Each specialization results in a child class, each with its own set of altered parameters. This creates a situation where all copies will continue to progress in an independent manner. Take the example of classes that handle monetary values and strings as a package. Their children classes may include one that handles the salary information of employees, while one may store the data of the attendees at a function.



**Figure 12.3** Example of Inheritance.

The following example gives a good idea of how inheritance is structured. In Figure 12.3, Vehicle is the superclass, which is also known as base class, and the others are subclasses of Vehicle. These subclasses inherit properties from the superclass Vehicle.

Remember, inheritance can be employed to set up a hierarchy of classes. Each addition only requires setting up new variables and data elements, while the basic set remains available to all the inherited classes. Inheritance simplifies the code and eliminates the instances where we may need to create the same data objects multiple times.

Now, let us see another example, where we create the basic class of a Tutor. We then create various extensions of the class to represent different tutors, like having a ScienceTutor and a LanguageTutor. These two extensions can be created as inherited classes, and only contain new variables that are important for a particular subclass extension. Here is a typical Java example for this situation:

```
package java11.fundamentals.chapter12;
class Tutor {
    String designate = "Tutor";
    String academyName = "NewAcademy";
    void performs() {
        System.out.println("Tutoring");
    }
}
public class ScienceTutor extends Tutor {
    String subject = "Science";
    public static void main(String args[]) {
        ScienceTutor obj = new ScienceTutor();
        System.out.println(obj.academyName);
        System.out.println(obj.designate);
        System.out.println(obj.subject);
        obj.performs();
    }
}
```

The output of the above program is shown below.



```
NewAcademy
Tutor
Science
Tutorial
```

This is an example where we first define the Tutor class that has two initial variables, which is the name designation and the academy name. Each inherited class can then add its own variables that will only be initiated for the members of that class and will not hold value for other objects of the Tutor class. However, we can block a subclass from using the members and methods of the original class if we define them as “private” during their initial declaration.

We can use the same technique to implement a set of well-defined classes that are present in a single hierarchy. This may not hold many levels when discussing people, but it is possible to have several subclasses when creating a complex Java program to imitate a complicated task.

It is possible to have different types of interference in Java. The primary inheritance is the single instance where there is a parent class, which gives rise to a child class. This is performed by extending a class to another class. This can be simply termed for two classes A and B, as B extends to A. However, it is also possible to implement multiple levels and create a large hierarchy, where each parent class extends a child and this occurs at least twice.

It is also possible to build a complete hierarchy, where multiple children classes are present and each has a single parent class. This is an excellent way of implementing functions, where we may have a single class structure that gives rise to objects which may also belong to other categories. Another type of inheritance is the seldom used multiple inheritance, where a single class may belong to multiple classes. Fortunately, this function is not available in Java 11.

Java employs the use of constructors when creating subclasses. It automatically creates inheritance, where all objects are constructed from the top to down method. It is possible to use a specific superclass constructor by employing “super” as the keyword. Remember, only one class can be termed as the superclass when creating a hierachal setting class objects.

### 12.2.3.1 Advantages of Inheritance

There are several advantages of inheritance, which we have already described. Here, we summarize the primary benefits of the OOP principle of inheritance, especially as they are available when employed in the programming environment of Java.

1. It allows us to derive further classes. This creates a reusable model where we never change the existing classes, which improves the software development time required for program executions.
2. The derived classes generate an extended set of properties, which ensures that programmers create dominant objects that are fully capable of performing the required independent tasks with loose connections to each other.
3. Base classes can give rise to multiple derived classes, creating a dynamic hierarchy capable of providing excellent functionality under different conditions.
4. It is possible to create complex inheritance, which offers the benefits of having all the properties present in the base classes. (Not possible in Java.)
5. All common code belongs to the superclass and only needs to be executed and compiled once during a program operation.
6. It is possible to override methods, which is excellent for defining empty method definitions in base classes and then overriding them, according to the specific use cases.
7. It is possible to optimize the code and arrange the required functionality in an enhanced manner. This ensures that the program code is effective and provides better throughput results.



Can a class have multiple inheritances?

### 12.2.4 Polymorphism

The last principle of object-oriented programming is harder to explain than the previous three principles. It provides the concept that it is possible to have objects that have the same hierachal position, but behave differently on receiving the same message. Their behavior is different in terms of producing an output when intrigued by a calling code. Figure 12.4 gives an example of polymorphism by showing how one interface can be used by various classes so they can have their own implementations. Figure 12.5 shows how methods can be implemented differently.

Polymorphism works by altering the way objects behave to the calling code. This means that they will carry out their functions in a variety of ways, based on their particular nature. This is a concept that provides an advanced way of making use of objects that are present in a language such as Java. This complex topic is used in high-level programming, but we will make sure that we give some examples in this chapter that will help you understand this important concept better.



**Figure 12.4** Example of an interface with different implementation classes.



**Figure 12.5** Example of Polymorphism by showing different implementations of an interface.

A better understanding is possible when we define polymorphism as the capability of modifying some methods of an object through overriding. This means that particular objects will serve as subclass instruments of their parent class, when they are showing different behavior when employed using the same calling code and instructions. In other words, objects of the subclasses may have different behavior from the objects of the other subclasses derived from the same superclass. The methods of all the subclasses may return different results.

A good example in this regard is to think of a class that defines big cats. We know that there are various big cats out there, and each has modified methods that they use to stalk and catch prey. Now, the calling code is in the form of a command that asks a cat to catch their prey. Each cat (say, lion or jaguar) will use modified methods to carry out this task. In this manner, the same object works as belonging to different derived classes.

We can define two types of polymorphism, according to its implementation. These two types are overriding and overloading. In this section, we will get introduced to these concepts.

- Overriding:** Here, the compiler is responsible for selecting the method that will be executed after the call. The decision occurs each time the code is compiled. This is also termed as *run-time polymorphism*, because the object takes different shapes during the program execution.
- Overloading:** This is when the specific method that is used by the object is determined by the dynamic position and type arrangement of the object. This is defined not during the execution, but is fixed at the time of program compilation, which occurs prior to execution having fixed behavior. It is also termed as *compile-time polymorphism*.

Polymorphism certainly remains an important principle, and it is one that truly separates OOP languages from functional languages that have a fixed code behavior. Take the example of a class which has a method of animal sound. We implement it within a class of AnimalProperties. Now, we know that each animal has a distinct sound. We want to ensure that the method of `animalSound()` returns a different result that depends on which element we are processing.

We will present two kinds of examples according to the two types of polymorphism that we have described in this section. Here is the first type of implementation:

```

package java11.fundamentals.chapter12;
public class AnimalProperties {
    public void sound() {
        System.out.println("This is top level animal class sound");
    }
}

```

Here are examples of how the same object behaves according to different object behaviors, using runtime polymorphism:

```
package java11.fundamentals.chapter12;
public class Cat extends AnimalProperties {
    @Override
    public void sound() {
        System.out.println("Meooww");
    }
    public static void main(String args[]) {
        AnimalProperties anmObj = new Cat();
        anmObj.sound();
    }
}
```

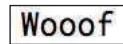
The above example produces the following result. Since we have overridden sound method, it will use the Cat class sound method to generate sound for Cat object.



Here is another way in which the same code would run differently:

```
package java11.fundamentals.chapter12;
public class Dog extends AnimalProperties {
    @Override
    public void sound() {
        System.out.println("Wooof");
    }
    public static void main(String args[]) {
        AnimalProperties anmObj = new Dog();
        anmObj.sound();
    }
}
```

The above program returns the following result.



The first runtime instance will print “Meooww” on the console, while the second one will present “Wooof”. Each behavior is different, although the same class is employed but is designed to have methods that show different behavior according to the required object.

The second example for polymorphism employs the compile time method. This is a technique, where a programmer overloads the required methods to produce distinguished responses. The control for this polymorphism is produced by using the arguments that we are passing to our calling methods. Here is a typical example:

```

package java11.fundamentals.chapter12;
class OverloadingMethod {
    void printOutput(int a) {
        System.out.println("the first number is: " + a);
    }
    void printOutput(int a, int b) {
        System.out.println("The two integers are: " + a + " and " + b);
    }
    double printOutput(double a) {
        System.out.println("The double number is: " + a);
        return a * a;
    }
}
public class OverloadingDemo {
    public static void main(String args[]) {
        double results;
        OverloadingMethod omObj = new OverloadingMethod();
        omObj.printOutput(20);
        omObj.printOutput(20, 30);
        results = omObj.printOutput(2.5);
        System.out.println("The multiplication results is: " + results);
    }
}

```

The above program gives following output.

```

the first number is: 20
The two integers are: 20 and 30
The double number is: 2.5
The multiplication results is: 6.25
|
```

This is an excellent example where we use a single object and keep overloading it in the next instance. The first method of `printOutput()` that runs outputs a single parameter of integer type, which is 20 in our example. The next instance of the method `printOutput()` then takes two integers as arguments, which we set up as 20 and 30. The third instance is when the same object performs a simple multiplication of finding the square of the argument, which is 2.5. The resulting answer is 6.25, which is displayed on the console.

This type of polymorphism is termed as *static polymorphism*. This is because all the shapes that an object can take are already well-defined and take their place during the compilation. There are multiple definitions of the same method. A particular definition is selected based on the argument that we pass on to the method each time we make a call. This means that the method will become static, and will always follow our defined parameters.

This type is certainly great for producing better control over the direction of the methods that can have different instances. A good example is that of a simple calculator, which we only need to produce a fixed quantity of functions. It would be an ideal polymorphic presentation. However, there are certainly other situations as well; we must ensure that the program can behave dynamically to pick out the best course of action.

Dynamic polymorphism ensures that the problem of overriding is only carried out at the runtime stage of the program. This is excellent for controlling methods that may be implemented by using a hierarchy of classes, from our earlier explained principle of inheritance. Let us take the example where we have two classes. The parent class is termed as class First, and the child class is termed as class Second.

There is a method in the child class, which is designed to override the method – say, `exampleMethod()` – that belongs to the parent class. When we assign the child class object in such an example for reference, then this is used to determine which kind of object would be produced at runtime. This means that the type of created object will actually control which version of the same method is called in the Java program.

Remember, it is not important whether the object is now defined as the parent or in the child class. The method is selected based on the specification given to the new instance of the calling class, which will control the flow.

```

package java11.fundamentals.chapter12;
class First {
    public void exampleMethod() {
        System.out.println("This method is to be overridden");
    }
}
public class Second extends First {
    public void exampleMethod() {
        System.out.println("Now overriding the method");
    }
    public static void main(String args[]) {
        First obj = new Second();
        obj.exampleMethod();
    }
}

```

The above program produces the following output:

**Now overriding the method**

We have created an object that even though it belongs to the first class, uses the construction type of the second class. This will override the initial method and we will get the console out of “Now overriding the method” accordingly. We can mix and match these override situations, but the runtime execution will always select the method according to the type of object that it identifies from the constructor of the object.

#### QUICK CHALLENGE

Think of any other real-life example of polymorphism and write a program for the same.

#### 12.2.4.1 Advantages of Polymorphism

There are several benefits of the principle of polymorphism. It ensures that programmers can use smarter code design schemes and achieve the following advantages:

1. It ensures that programmers can first fully test and finalize their code before implementing it in a variety of ways. This way, it is possible to always use consistent elements in the final program.
2. Developers do not need to take care of the naming schemes, as polymorphism ensures that the same name can represent different data instances, such as int, double, and other available options in the OOP language.
3. It reduces the present coupling, ensuring that we can create flexible program objects. (We have discussed coupling earlier.)
4. It improves the code efficiency, when a program becomes long with complex functions placed within it, as it ensures that several instances and situations can be adequately handled.
5. Closely related operations become possible by using method overloading, where we can use the same name to designate different methods, each with their own set of parameters.
6. It allows the use of different constructors that can initialize class objects. This ensures that we have program flexibility, with access to multiple initializations.
7. It is carried out during inheritance principle application as well, as general definitions of a superclass can be employed by various objects that add their specific definitions, using the overriding of the available class methods that are not instantiated properly in the superclass.
8. It is excellent for reducing recompilation needs, by ensuring that even sections of a class can have a reusable structure, while the altering requirements may be achieved with the use of polymorphism.



Are there any disadvantages of using polymorphism?

## 12.3 | Object-Oriented Programming Principles in Application

The OOP principles are important in programming since they offer several advantages. This method is perfect for applying on large programming projects. It ensures that we create reusable code and eliminate redundant elements in our programs. The combining of the running code with the data is perfect for creating self-contained modules that can then employ the advantages offered by modern computers.

OOP principles are excellent for several applications. The most important application is in creating dynamic applications that need updating every few weeks. This is often the case for security programs and ones that help in financial decisions. OOP programs are easily modified as they can have a modular structure. The chances of mistakenly entering wrong logic are also slim in OOP languages. While Java may not be a complete OOP language, it does offer the benefits that are associated with OOP principles.

Another primary application for OOP principles is in creating large-scale programs. These are complex programs that take on many inputs and then perform a number of calculations to produce the desired results. Creating objects allows programmers to set up distinct sections and produce privacy levels. All elements that must remain stable are defined in a private state, while ones that can benefit from an update are set up in a public setting.

Java and other OOP languages are great for creating server/client programs. These programs need efficiency and often require large memory and processing resources. These languages are excellent at creating real-time application solutions, which need to implement reactive programming elements. Parallel programming is best performed in OOP languages that can set up differentiated tasks and define a strategy that provides the ideal scenario for using concurrency during executions.

Artificial intelligence is another avenue for creating OOP programs. Here, we need to make automated improvements to ensure that machine learning becomes a possibility. Any programming application where efficiency and the ability to create smaller modules are required will be best served by using OOP principles and creating reactive programs. These are easily improved while ensuring that the primary legacy elements remain secure and invisible to programmers that may add further functionality.

It is also possible to have bugs in your large programs. OOP languages are excellent for performing the required debugging and the improvement of your code. The use of microbenchmarks can help you identify the performance of various program approaches, ensuring that you avoid logical errors and create a final program version, which is free from a buggy performance.

Since objects often follow inheritance and polymorphism, any errors are easily found. They can often occur due to the misuse of the objects and will seldom require the actual changing in the private object structure, which is another benefit of OOP. With these applications in mind, beginning programs should learn OOP principles and apply them in Java, which also offers primitives along with objects.



Can you use both interface and abstract on a class?

### 12.3.1 More About Objects

Objects are the primary unit of all OOP languages. They serve as elements that are individual and fully capable of communicating with each other. They can also form hierachal structures and implement logical decisions based on the states and the returns of different operations. Objects are a collection of data types, elements, and their controlling and governing code.

Objects also follow the inheritance principle, where parent objects can give rise to various child objects. The child objects will always be the extensions and contain additional elements that provide a greater functionality over the basic object function and data structure. They are also excellent in terms of only holding the exact information required for their successful performance.

There are several benefits of using objects to employ OOP principles. They allow you to use programming in a way similar to the real-life decisions and understanding of concepts. We define each object by using specific attributes, while also defining a role for it. Programming languages mimic this behavior and provide the functionality of using logic and modularity for designing complex functions during software development.

It allows for realistic control designs that follow the ideal use examples. If we have a gear change system in a car, it always follows the available gears and cannot go beyond the available ratios that are provided by the components present in the transmission. We can implement the same behavior in Java objects, where they only behave in a predictable manner, by only following and taking on values that we have allowed in our object definitions.

Another advantage of using objects in OOP is the use of modularity. We can create independent objects which have the ability to provide a dedicated functionality in a variety of settings. The information that defines the objects is hidden from any instance of the calling codes. Therefore, it works perfectly for creating modules that can interact with each other but not produce any undesirable effects on each other during program executions.

Remember, Java objects work under the system of classes. It provides them an external structure. All Java libraries are a collection of classes, with several methods provide functionality by implementing reusable code, which has already been prepared by an expert for optimized Java functionality.

### 12.3.2 Classes and Object-Oriented Programming Principles

Classes are important in OOP languages like Java. This is because they allow programmers to use the same kind of logic that we use in the real world to define objects and classify them based on their properties and the function that they can perform.

Let us take the example of cars. There are different types of cars, but all of them can be defined as modes of transportation that use fuel to power locomotion. Your car can be simply defined as one of the instances of the class of cars. It may have certain properties and characteristics that are present in other cars. However, it will never show a property that makes it functionally different from another car. It can have additional behavior though, such as inclusion of a refrigerator and additional entertainment equipment. Figure 12.6 shows the difference between Classes and Objects.

The OOP principles are best acknowledged and used in programming when we employ blueprints. Classes work by providing an important structure, which then allows programmers to build up additional facility and produce improved programming results.

Class definitions only provide the structure and do not have execution element within them. This is provided by the `main()` method, which actually carries out the execution in any Java program. Once you set up these classes that define behavior which may be extended using additional values, it is possible to use the objects that embody the OOP principles that we have described above.

Classes allow the use of the principles of defining every functionality in the form of object instances. With the application of other additional methods, objects can then be used in a modular manner for the intended result. However, programming languages such as Java can go beyond the simple OOP applications and provide additional facilities.

#### QUICK CHALLENGE

Create a food class and show all the possible child classes of that type. For example, fruits, vegetables, etc.

## 12.4 | Understanding an Interface

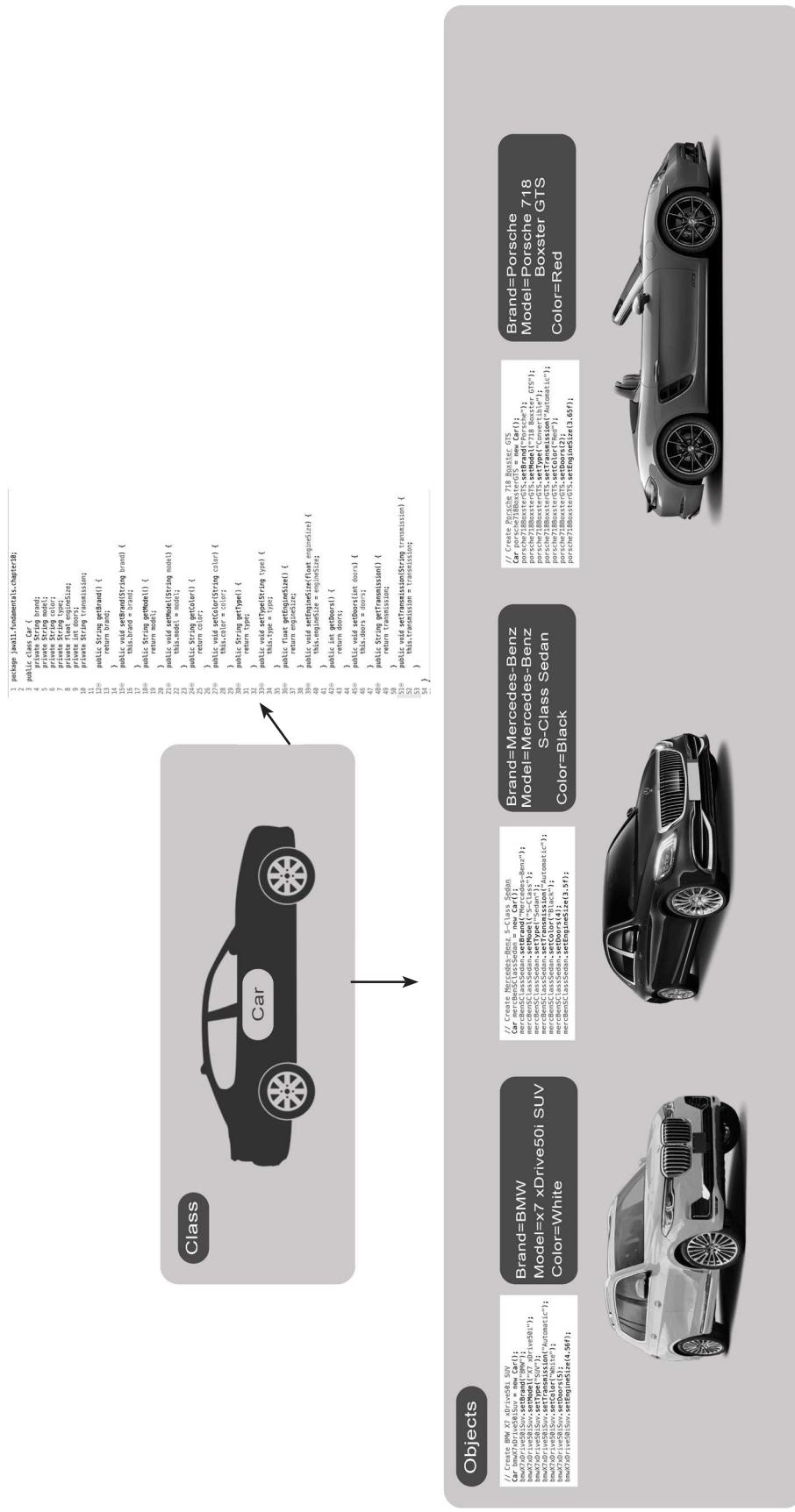
Objects need to interact with the environment around them in the programming application to ensure that they can perform the required interaction. They use methods that expose them and allow them to offer OOP functionality. Objects in Java use the interface to deliver their messages and interact with the running program code. An interface, therefore, is a set of methods which have empty bodies that certain objects can use for their particular function.

A language like Java uses the implementation of different interface elements to show a dedicated and definable behavior. The keyword of “implements” allows the creation of a class object which will implement the empty methods, ensuring that they are employed to allow the object to have a certain formal behavior.

The creation of an interface ensures that the program will only compile when the object uses all the methods that are present in that particular instance. However, this means that they are all successfully defined within that interface element for the successful compilation of the class during a program execution.

Interfaces further simplify the use of classes and objects in Java. An interface provides the answer to the functions of a particular code. Let us again consider the example of a car. A driver can use the steering to ensure that the car remains on the intended part. The steering wheel provides an interface here to use the transparent functionality.

However, the driver may not know at all how the steering process actually works, in terms of the drive axles, power steering mechanism, and the behavior of other mechanical components. The implementation in Java describes these details, which in most applications are redundant and are not required by the user. Interfaces are great to set up when a certain program must deliver a functional process, which is to be used by a normal software program user. This makes interfaces important, especially when employed with Java classes.



**Figure 12.6** Example of Class and Objects.

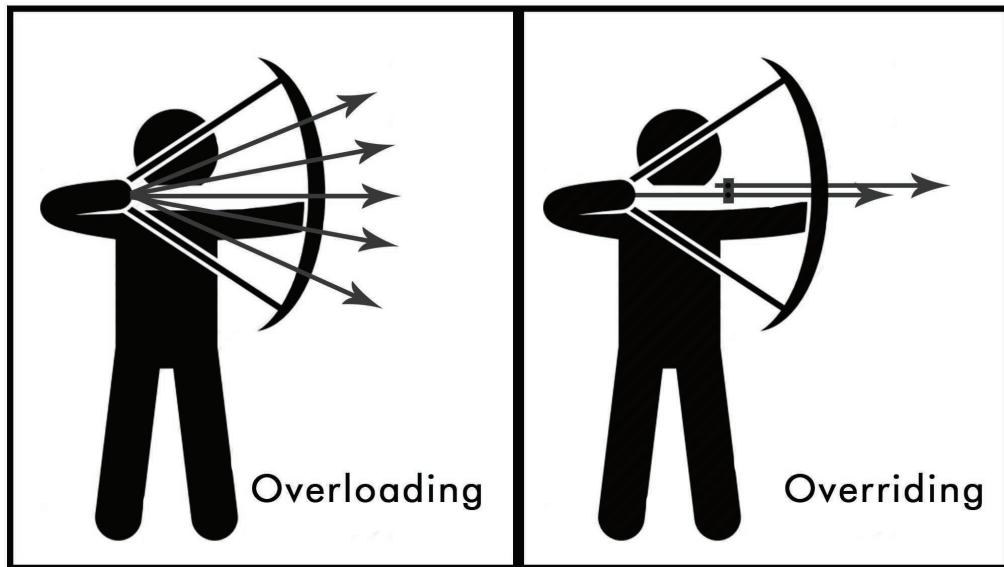


Can an interface extend multiple interfaces?



## 12.5 | Overriding and Overloading

There are two important concepts in Java known as overriding and overloading (Figure 12.7). We will explore both these concepts in this section.



**Figure 12.7** Explanation of Overloading and Overriding.

### 12.5.1 Overriding

Overriding is a process often used in Java programs. You will often create programs where you want to set up methods that you can alter when dealing with different objects or classes that are derived from a superclass. The simplest definition of overriding is that an instance method that occurs in the superclass will override the main method described in the superclass.

One way of using this technique is by writing “@override” annotation before the method. The use of this annotation produces an error from the compiler if it does not find the method first present in the superclass in a dynamic manner. In fact, the compiler actively finds that the method present in the superclass has been overridden in a subclass ahead. If it does not see an overridden implementation of a method definition, it will generate a compiling error.

Remember, overriding is often confused with hiding a static method. Overriding is different since it invokes the method which is described in the subclass. On the other hand, hidden methods can be invoked both from the superclass or the subclass.

In fact, overriding methods in Java offer an important example of an OOP principle. It allows the use of polymorphism. The overriding produces runtime polymorphism, where one object can have different conditions because of the different ways in which the methods can be implemented in the subclass.

Method overriding is easy to use by learning some basic principles.

1. The first rule is obvious, which is to write the exact name of the method as present in the superclass.
2. The second rule is to have the same number of parameters. Remember, a method changes how it behaves but this does not mean that the parameter arrangement can be altered.
3. The third rule for overriding method is the presence of a relationship. The methods can be overridden if they are present in a subclass, where they have an inheritance relationship in the form of the “Is A” model.

Here, we present a simple example of how method override will be employed in the syntax of a Java program. In the program, we present transportation as our superclass and describe the instance of a truck as the subclass, which invokes an overridden method.

```
package java11.fundamentals.chapter12;
public class Transport {
    void run() {
        System.out.println("We use transport vehicles for movement.");
    }
    public static void main(String[] args) {
        Truck obj = new Truck();
        obj.run();
    }
}
class Truck extends Transport {
    void run() {
        System.out.println("We use trucks for transporting loads.");
    }
}
```

This is a simple program, where the `run()` method has two different versions. When we run this method in the `main()`, we employ its second instance, when it will print the message of “We use trucks for transporting loads”.

**We use trucks for transporting loads.**

This creates a condition where the same program is capable of running different responses based on the OOP principle of polymorphism.

Overriding is important when we have similar objects that we can place in a parent class, but the parameters or formulas that we want to employ in our program may differ. Take the example of banking calculations that will always use the formula, but there may be a change required in the interest rate to ensure that the calculation occurs according to the needs of the specific program object catering to a specific bank.

Remember, we override methods that are dynamic in nature. Dynamic methods are the ones that are called from the objects of a class, rather than the ones that are only described in the parent class. There is another method that is functional in other instances of polymorphism, which is called overloading.

## 12.5.2 Overloading

The other technique available for changing the methods present in a Java program and then using them for our specific function is the method of overloading. This is a feature where we can use the same name for multiple methods, where the method employed in the program depends on the arguments that we pass during a method’s use. This is a technique where we can use multiple constructors that work on different argument conditions.

This syntax structure available in Java is just like using a constructor to develop the overloading function to have different class extensions. Overloading works by reading the details each time a method is employed in the main program. Take a simple example where we have a multiplication method. This method has two argument instances.

One instance describes the use of two numbers, while the second one is designed to calculate the multiplication of three numbers. Here are the two ways in which they can be run in the code, like having forms of `multi(int num1, int num2)` and `multi(int num1, int num2, int num3)`. We can clearly observe that the only difference between the two method calls is that their argument list is distinct from one another, apart from the actual variables that may be present in the calls.

This type of changing the method is the example of static polymorphism. This is because the type of object required is prepared at the compilation time, when the parameters from a method call are read to understand the required shape of a Java object. Here is an example that describes the use of overloading in Java syntax:

```

package java11.fundamentals.chapter12;
class OverloadMethods {
    public void displaying(char a) {
        System.out.println(a);
    }
    public void displaying(char a, int num) {
        System.out.println(a + " and " + num);
    }
}
public class OverloadExample {
    public static void main(String[] args) {
        OverloadMethods obj = new OverloadMethods();
        obj.displaying('A');
        obj.displaying('A', 100);
    }
}

```

The output of this program is in the form of two lines.

A  
A and 100

This is a program which describes the syntax settings of the overload method functionality. We see two separate definitions of the `displaying()` method which both occur in the class initialization. We then create an object of our class, which then employs the variable two times. We only pass a single argument for the first time, which is A. This shows that we are looking for the method which only takes a single character datatype as an argument.

We then employ the method once again, but with two arguments. This automatically lets the compiler know that we are looking for overloading the method with the second definition, which takes as argument one character and one integer. The program automatically now calls this specific method definition and runs the program seamlessly. This is an ideal way of creating programs where you control the number of methods that you have.

You can tweak their definitions whenever you want to perform tasks which are quite similar, by simply presenting different ones in the class definition and separating them with the use of separate set of method arguments for specific use. The arrangement of the arguments, and the type of arguments are all important for providing the directions for the compiler to perform the method overloading. Each time the method definition, which exactly replicates the method call, will be employed in an active Java program.

However, there are scenarios where the Java language syntax is important for controlling the results of method overloading. The datatypes do not have to be fixed when calling the method. The concept of type promotion is employed. This is easily understood with the help of the following example.

If you have a method call with a floating number but the available method definitions only have them for a double integer, then the number will be changed into a double integer. However, this promotion only occurs when the compiler does not find an exact method definition for passing a call in the program code. The rules that govern type promotions in method overloading are:

1. byte to short to int to long
2. float to double
3. int to long to float to double
4. long to float to double
5. short to int to long

Always remember that since method overloading is controlled by having specific and independent set of definition arguments. The compiler will throw an error exception if it finds multiple method definitions that simply take similar arguments. Table 12.1 shows the difference between overloading and overriding methods.

**Table 12.1** Overloading vs Overriding

	Overloaded Method	Overridden Method
<b>Argument(s)</b>	Arguments must change in overloaded method	Arguments must not change
<b>Return Type</b>	Return type can change	Return cannot change. The only exception is covariant returns. A covariant return type can be replaced by a “narrower” type in case of overridden method in subclass.
<b>Access</b>	Access modifier can change	Access modifier can be less restrictive but must not be more restrictive. For example, a superclass having a private method can be defined as protected or public in the subclass. But any public method in superclass cannot be declared as private.
<b>Exception</b>	Exception declaration in the subclass can change	Exception declaration can be removed or reduced but must not throw new or wider checked exceptions.
<b>Invocation</b>	At compile time, reference type determines which overloaded version should be selected.	At runtime, object type determines which method should be overridden.

Overloading and overriding are two very important points in OOP. You will be using these two often in your development career. Hence, it is important to master these two concepts.

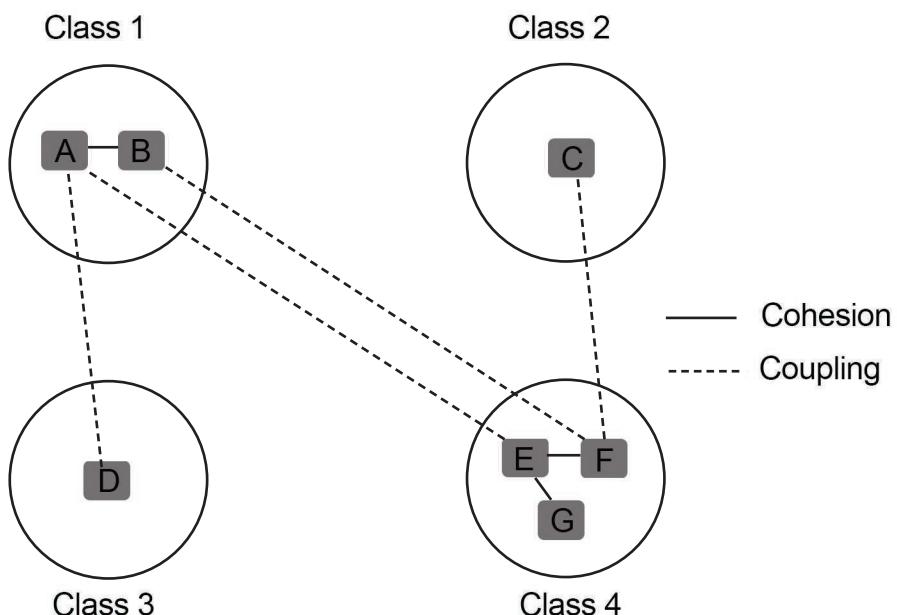
**QUICK CHALLENGE**

Think of at least five examples where you can apply overloading and overriding principles.

## 12.6 | Coupling and Cohesion



You will be able to deliver quality code and solve complex problems if you can embrace these two OOP design principles – coupling and cohesion. You should always aim for loose coupling and high cohesion. Let us take a look at what these two terms really mean.



## 12.6.1 Coupling

Coupling can be defined as the level in which one class has knowledge about another class. If they do not have any common shared knowledge; that is, if one class named A only knows class B via its exposed interface, these two classes can be termed as loosely coupled. On the other hand, a bad design could be if class A relies on parts of class B that are not part of class B's interface. This type of coupling is known as tight coupling, which you must avoid in your design.

## 12.6.2 Cohesion

Cohesion has to do with the level in which a class is self-contained. A single, well-focused purpose makes the class highly cohesive. There are two major benefits of high cohesive classes.

1. The classes that are highly cohesive are easier to maintain.
2. The well-focused nature of the classes makes them more reusable.

## 12.7 | Implementation in Java

Java is an excellent programming language and contains many Application Programming Interfaces (APIs), graphics facilities, and other options that are especially great at implementing the OOP functionality. It is a language which now offers the read-eval-print-loop (REPL) system, allowing for quick compiling of code to see how it will behave in a realistic environment.

The core Java features adequately cover all important OOP concepts, which include encapsulation, inheritance, and polymorphism.

Java also offers powerful factory methods that provide much more functional set of objects and classes. This comes with strong interfaces, which can describe any situation and ensure that it is possible to explicitly use code in a manner which ensures that no programmer will be able to use it in a wrong way.

Java is fundamentally a powerful language where you can use all the empowering principles of object-oriented paradigm. However, it still allows you to stay away from situations where you may feel limited by the creation of subclasses and the inheritance implementation structures. It usually imports interfaces and therefore, reduces the instances where we must implement classes in an unintended manner, which often creates further complexities in a Java program.

Creating the interface is important in most Java programs. Interfaces can reduce the number of methods and code that you need to carry out certain functionality. They are often employed with the `@override` annotation, which describes that the method in use is overriding its definitive function, whether it is described from a particular class or a specific interface. Remember, a failed compilation will never change a method in Java.



How many methods can a class have?

## 12.7.1 Objects Work as Solutions

Java is an excellent language because it allows the use of objects as solutions that resolve well-defined problems through the described parameters. The functions required from a program describe the problem, and then we end using multiple objects and interfaces to carry out the same functionality by using abstraction and employing inheritance at various levels.

Another way in which this situation works is by carrying out problem solving using the available states of objects. We set up the behavior of these attributes in a manner that the presence of the required data ultimately provides the situation, and where the ideal solution to the required solution is identified and achieved.

Most Java programs are created following a specific pattern. First, an industry expert works out the logical solution to a problem. This expert may know about programming or simply provide a human language algorithm or a set of required logical steps. Then, Java program developers use this available information to create an inheritance structure and set up the parameters required to deliver the intended functions through OOP principles.

Java offers multiple avenues for using objects. It allows setting up relationships which are ideal for eliminating code duplications while still ensuring the smart use of available programming avenues and advantages. The major advantage offered by languages such as Java when they first appeared on the horizon was to finally use a programming paradigm, where it was possible to eliminate the repeating code and therefore, create the fountain model of programming.

Object-oriented languages allow programmers to focus on the complex programs, while leaving the emphasis that was often placed at optimizing the code. Java is a language which uses the passing of messages for communicating with the different objects and creating a paradigm where the specifications of the message remain hidden from a common developer using them in a program, thereby creating an efficient working model.

This gives rise to the attributes and the behavior structure, which is shown by class objects in all programming languages. The attributes of an object is set up the way it is defined, especially when using the language to define solutions. The behavior depends on the arguments which are passed to different objects and the methods defined in a class. They employ the various modes of solutions based on the program inputs and outputs.



How many objects can you create in one class?

## 12.7.2 Ideal Tips for Implementing Basic OOP Concepts

There are several tips which you can follow when implementing OOP principles in your Java programs. Remember, the primary objective of object-oriented languages is to offer swift programming without jeopardizing the security of the prepared programs. The following are the main tips for implementing basic OOP concepts:

1. The first tip is to never repeat the same code in a language like Java. This is a key OOP objective, since we declare all tasks in the form of our required functionality. We can always call the related code accordingly by dividing our data objects in terms of the tasks that we need in a specific program. To avoid this, you should always create a method for a code section that you intend to use multiple times. This will allow you to simply call that method and process the data accordingly in each instance.
2. Encapsulation is the key if you want to continue to improve and alter your code in the future. This is a possible by setting up private methods and variables. You can always change this access to a protected one, which will allow you to always copy the code without ever affecting its primary behavior.
3. Your code should always follow the principle of singular responsibility. Each class that you define and use must cater to a singular need. This means that it will always be easier to call and extend your program classes without affecting the program by a significant margin. Avoid the coupling of key program functions. You can create class objects that combine results from different classes to perform the primary function, but avoid mixing multiple responsibilities in unique code units.
4. Another trick to follow is the use of the open-closed design. You must always keep your classes closed for any modification. This provides data security and ensures that you are following the OOP principles of abstraction and encapsulation. However, your individual methods and specific classes must remain open for further extension in terms of functionality.

This is a great practice to ensure that you can hold on to your successful code, while still allowing your program to embrace new elements that result in improved functionality and scalable program behavior.

You may be wondering how you can achieve the required functionality, if your program is complex and often needs to process large sections of calculations and data elements. The ideal trick to this is to use as many objects that you need. This is the main principle of using an OOP language to your advantage. You create specific instances, call the same methods from the class definition, get the particular task done, and then move on to run other objects in exactly the same manner.

Remember, you do not have to create smart objects that perform complex functions. Use multiple ones and use them within each other for the intended functionality that should always have a singular nature. This allows programmers to truly employ the benefits of using the OOP principles to create an efficient code for any intended function.

5. Another tip is to judiciously use the interfaces that we have described above as special elements that help in OOP implementation. They are by design the helping tools for creating specific objects; use them in a contractual manner. Interfaces should be set up to give hints about the methods that should always be implemented in your source code.

You can work in teams and divide the tasks that you must perform. This is possible if you use specific interfaces that force your employees or coworkers to ensure that they utilize the available methods properly. This creates a coherent code and ensures that all instances of a particular interface always allow the presentation of a unified set of methods.

6. Another ideal tip is the use of dependency injection. This is an excellent technique where the use of a static method or an object is employed for creating the dependencies and important needs of other objects. The dependency is also created using an inject, in which the injection is carried out using the transfer of properties using a client object.

This is an excellent functionality which allows Java developers to implement any type of testing. This technique forces programmers to create database objects which are easier to control using external functionality. This is perfect for testing and ensuring that you can fully obtain the benefits of programming in an OOP language.

7. Another key tip is to compose the inheritance of objects in the right manner. We can use these relationships and the concepts of objects being connected with each other to simply the OOP programming needs. When taken in the context of Java, this means that we should create classes which are simple and can always be extended. This is possible by the strong setting of inheritance and defining the functionality of each in a careful manner.

Remember, an object may be a sub-element of a class, while still having a specific relationship with another object. A good example is that of an apple. An apple can be an object of the fruit class. It can then be described to have hard seeds. The distinction that we need here is that the main class must always be the fruit, as otherwise it will be difficult to define instances where a different set of characteristics is required.

8. Another tip is to create various classes that only connect to each other in a loose manner. If you create objects that are strongly attached to each other, it makes it extremely difficult to reuse your code to its full potential. Create situations where you perform inheritance, but keep the structure flexible enough to always implement additional functionality and the required attributes.

You may have to first study and gather more information about the use of loose classes since the normal concept of inheritances to make a strong attachment between the objects. Remember, creating a code which is flexible and reused plenty of times is far better than creating a strong code which is perfect for use in a typical situation. Consequently, it becomes a burden when used elsewhere in the program because of its dependence on other objects for its optimal functions.

## 12.8 | Future of Object-Oriented Programming

A key element that we must discuss in this chapter is the future of OOP programming. As we move towards machine learning and the creation of smart code capable of carrying our self-improvements, it is important to consider the future of even the most successful paradigms of today. Here, we present arguments from both sides. We present the bright future ahead as well as describe some pitfalls that are on the horizon for the OOP world.

We first start by presenting a few downfalls. We believe that by knowing them, you can reduce the instances where you face these problems and learn to use OOP programming in the right way, where you become part of the future of OOP programming.

### 12.8.1 The Downfalls

Inheritance is the key for OOP programming languages. However, it may not be so easy to use in modern, complex programs where it is possible to create large hierachal structures that will all need to be copied in your next program, if you aim to use a child class. This is especially difficult when your class objects also refer to other objects, because you will then require the class hierarchy from all involved objects.

Programmers describe this as the banana problem. If you want to copy a banana from a program code, you will end up copying the banana, its tree, its environment, and the whole jungle just to ensure code consistency through the class structure. This problem appears because of the implicit environment, and computer scientists are already looking at how to control and manage the problems that may appear because of extensive use of class hierarchy structures.

Another problem is produced by encapsulation, which often produces a complex situation during code compilation. A code reference is passed, where it may be required to create a complete and detailed clone object to produce the required level of functionality. Polymorphism is described as an OOP principle, but it can be easily implemented in other ways. In fact, Java allows you to use specific interfaces that produce the same functionality but in a more organized manner.

However, these are common issues and you can work around these problems. The future of the OOP world remains bright, as we describe in the following subsection.

### 12.8.2 Future Applications

There are various applications that are ripe for use in the future, and are already on the horizon. Take the example of web APIs, which are being prepared with the use of OOP languages such as Java. However, these functions are better prepared using functional programming languages, and current OOP languages also create a shell that produces the same functionality.

One of the best applications of OOP is in the field of user interfaces. User interface (UI) can produce situations which are uncalled for and cannot be resolved by having fixed structures. This is perfect for an OOP language because it is an excellent approach to build the dynamic structure. OOP languages are also great at producing frameworks, as there are several communities out there which provide support for platforms such as the Java Community that supports all Java functions.

Another application is in the field of cloud-based software solutions. Software as a Service (SaaS) is a particular application where OOP languages offer a winning proposition. OOP languages are perfect for creating solutions that can take advantage of the available resources, providing high processing functionality, and using optimum memory resources to produce high impact applications.

There is a strong shift to the cloud and modern data centers that are opening all the time to provide the intended functionality. These centers offer a shared pool of resources and want to provide the ideal service by empowering their hardware resources through the addition of excellent software applications, which can be best prepared using OOP principles, the languages that offer reusable code, and the ability to offer constant updates.

OOP languages can prepare software solutions that can save money for all individuals. Modern data centers need solutions for client/server interfaces, client-side applications, and programs that allow for improved division of the available sources. This is all possible by using a modern OOP platform such as Java. It will continue to get better with a more modernized version already in the pipeline, which helps to provide further support towards the creation of cloud applications.

It is also possible to use languages such as Java with complex platforms and other working elements to create effective situations where it is possible to come up with the ideal results. OOP languages are becoming more powerful, as their discrepancies are reduced by modern computational resources. With the possibility of endless heap size available to the programs, you can produce complex inheritance and create programs that take the full advantage of OOP principles in all facets of the application.

## 12.9 | Understanding the World

The OOP programming is engaging since it allows developers to equate development ideas to the real-world scenarios and find comparative solutions. There are scenarios like “The Diamond Problem” which are difficult to produce in the OOP environment. However, they represent exceptional cases, where some languages like JavaScript offer a way out for programmers. The Diamond Problem, also known as “Deadly Diamond of Death”, is a multiple inheritance problem. This problem occurs when two classes, say, Movable and Recliner inherit from Furniture, and class Chair inherits from both Movable and Recliner. Now say, equal method is called for Chair and there is no such method in the Chair class but it is in both Movable and Recliner classes. In this case, upon calling equal on Chair, which equal method should get called as Chair class inherits equal method from both Movable and Recliner? This is called “The Diamond Problem”.

Procedural and functional programming paradigms are also helpful. You need to understand when to use OOP to achieve the ideal performance. Using an object-oriented language allows a project manager to divide the development tasks and create sizable chunks of work, where everyone can pitch in without affecting the overall workflow.

A key problem associated with current solutions is that class hierarchies and code supporting elements can often require a large heap size for efficient solutions. With increase in the available resources, it is now easier to implement complex programs that make full use of the important OOP paradigms as independent objects and fully extendable basic classes.

Always remember that we should not force each problem to be resolved using OOP principles. Rather, the ideal way is to start your project management by describing the problem and then work out the possible logical solutions. This will allow you to come up with the relevant OOP ideas in most cases, where you may also learn that some programs may work better using procedural languages.

## Summary

---

OOP is one of several programming paradigms that are in use today. It started out as a programming discipline for creating mainframe computer programs and applications, and became popular gradually. The OOP structure is defined through the principles of encapsulation, abstraction, inheritance, and polymorphism. These principles are incorporated using different tools by various OOP programming languages. Java employs the system of classes and objects to apply these principles.

We have described each of these principles in depth in this chapter. We have presented examples and situations where the OOP paradigm is applied to resolve problems. Another thing that programmers must remember is that each OOP language has its own particular flavor. Some languages cannot be defined as purely object-oriented because they offer other avenues

of creating hierarchy and carrying out a task without creating a strong boundary wall between code implementation and its external behavior.

We also describe the presence of some OOP tools that Java provides. It cannot be termed as a purely OOP language, so we also discussed elements such as an interface, which is present in a different scenario from the implementation of a class object. We described how these principles are placed into action by working out realistic examples.

However, you should remember that this chapter serves as a basic introduction to the OOP world. We have provided some excellent pointers here that you can use to improve your understanding of this programming paradigm.

We completed the chapter by providing an overview of what the future holds for OOP practices. We have identified the key scenarios where OOP languages will continue to excel, while also marking the situations where you should understand the shortfalls of this programming practice.

In this chapter, we have learned answers to the main questions related to OOP:

1. What is object-oriented programming?
2. How does object-oriented programming help create better solutions?
3. What are the principles of object-oriented programming?
4. How do inheritance, polymorphism, encapsulation, and abstraction work?
5. What are the benefits of using abstract classes?
6. How can interface be used to add features to a class?

In Chapter 13, we will learn about generics and collections. We will explore various collection APIs like Map, Set, List, Queue, Collection, SortedSet, SortedMap. And then we will see the concrete implementation classes such as HashMap, Hashtable, TreeMap, LinkedHashMap, HashSet, LinkedHashSet, TreeSet, ArrayList, Vector, LinkedList, and PriorityQueue. We will also learn about Stream API for bulk data operation, forEach, forEachRemaining, removeIf, spliterator, replaceAll(), compute(), merge(), etc.

## Multiple-Choice Questions

---

1. Which one of the following specifiers applies only to the constructors?
  - (a) Public
  - (b) Protected
  - (c) Implicit
  - (d) Explicit
2. Which of the following is not a part of OOP?
  - (a) Multitasking
  - (b) Type Checking
  - (c) Polymorphism
  - (d) Information hiding
3. We use constructors for which of the following?
  - (a) Free memory
4. (b) Initializing a newly created object  
(c) Building a user interface  
(d) Creating a sub class
4. run-time polymorphism is known as \_\_\_\_\_.
  - (a) Overriding
  - (b) Overloading
  - (c) Dynamic Binding
  - (d) Coupling
5. What does OOPS stand for?
  - (a) Object-Oriented Programming System
  - (b) Object-Oriented Processing System
  - (c) Object-Oriented Programming Structure
  - (d) Object-Oriented Personal Structure

## Review Questions

---

1. What is an abstract class?
2. What is interface? How should we use it?
3. Can you use more than one interface on one particular class?
4. Give a real-life example of polymorphism.
5. How does encapsulation help developers?
6. What is inheritance? Give an example.
7. What is the difference between a class and an object?

## Exercises

---

1. Think of a real-life example of abstraction and draw a diagram. Explain the example.
2. Explain in detail the benefits of inheritance. What would have happened if OOP had missed this principle?

3. Write a program using Eclipse IDE that contains interface and abstract classes. Then create classes that implement these interfaces and extend this abstract class.
4. Explain the concept of overloading and overriding with examples.

## Project Idea

---

Traffic is a day-to-day problem for urban areas. There are various types of vehicles that make it difficult for cyclists and pedestrians. Plan out a software that will allow the city to divide into three types of roads – one for vehicles, one for

cyclists, and one for pedestrians. The rule is that a type cannot use the road that are not intended for its use. Make a detailed diagram for this problem.

## Recommended Readings

---

1. Brett McLaughlin, Gary Pollice, David West. 2006. *Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*. O'Reilly Media: Massachusetts
2. Grady Booch, Robert Maksimchuk, Michael Engle, Jim Conallen, Kelli Houston, Ph.D. Young Bobbi. 2007.
3. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional: Massachusetts
3. Lesson – Object-Oriented Programming Concepts: <https://docs.oracle.com/javase/tutorial/java/concepts/index.html>

# Generics and Collections

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Generic programming and its application in Java.
- How generics can be employed to improve program functionality and provide room for improvement in the programs.
- Object collections and the functionality that they can offer in Java.
- Important APIs and Java methods that help you use object collections for data manipulation.

### 13.1 | Introduction

Java is a powerful programming language that facilitates programmers by combining innovative object-oriented functions with legacy options. In this chapter, we focus on a largely ignored part of programming languages, which is the ability to deal with abstract algorithm structures that are later capable of providing the required level of support. Object collections also allow programmers to create and manipulate sets of objects that may offer extended functionality. We will discuss the generics and object collections that can be employed in Java.

This chapter presents a thorough discussion of the implementation classes, application program interface (API), and other options that are available for use in the language, such as mapping and collecting functions. This chapter stresses on the methods that describe the use of generics as well as the important collection schemes that are perfect for use with Java.

We will first establish the objectives of the chapter, then give brief descriptions of the terms of generics and collections in software engineering, and finally discuss the specific functions that you can employ in Java object collections and generic programming methods.

We aim to ensure that this chapter provides the required information for improving your generic programming applications and ensure that you take full advantage of an OOP language that provides object collections for use in complex programming applications.

### 13.2 | Generic Programming



Generic programming is a key concept in software engineering. It involves the use of algorithms that have the ability to create data types, which can be later specified by instantiating them with a particular set of object or type parameters. This approach has a respectable history and is designed to ensure that it is possible to provide functionality, which can reduce duplication in the program code. Multiple languages employ this functionality, including Java.

Generic programming concept is termed as parametric polymorphism in some programming languages, because it allows the language to have elements and code blocks that can have different forms based on the requirements of a program. Abstract program code is perfect for use in conditions where different programming patterns need to be employed from effectively the same form of code. Generics are also great to ensure that programming errors can be reduced by forcing programmers to use safer methods that allow them to stay away from logical errors.

Java employs generics as a programming facility, which is present in the type system identification in the language. It can be implemented as an object or method type, which is designed to cover operations on different data types, while ensuring that compiling errors are eliminated and a safety net is implemented that takes away logical errors.

Generics in Java have a strong structure which contains type parameters like the following. Variables provide the parameters, which ensures a structure of the class.

```
class class_name<T1, T2, ..., Tn> {}
```

A generic class is capable of declaring multiple type variables, such as Integer and String, in angle brackets (<>). See the following example:

```
class MyClass <Integer, String, Boolean> {}
```

The above code ensures that the class has the required set of parameter types such as Integer, String, and Boolean, which can cover any possible invocation of the class parameters during application.

A generic interface is one that contains type variables, which act as parameters and provide the same interface during the runtime phase of program execution. A generic method is similar to normal Java method, and it can type parameters that may either be following a class object or a Java interface. A generic constructor can work independently in Java apart from its class. This is possible because the Java collections framework is capable of dealing with constructors separately, which may have a parameter list of a generic class or the interface.

### 13.2.1 Benefits of Generics

Generics have their specific benefits, which suggest that programmers need to use them over other coding schemes. When employed with a programming language such as Java, it allows classes and interfaces to act as parameters and therefore, set up parameters that have greater usability. It is possible to use different inputs and reuse the same code to a greater benefit, which is the main appeal of employing generics.

Java code having generics contains other benefits. Let us take the example of employing data type checks, which occur at the time of compilation. Generics ensure that the compiler can pick up on any errors and generate an error code that describes the missing type safety. This problem is easy to identify and resolve. The same problem goes into the runtime phase if not generics is not used. It becomes a time-consuming exercise to detect, debug, and remove the logical errors that are present in the employed datatypes.

Another benefit of using generics is that it eliminates the use of casting. It is common for the compiler to cast datatypes when adding items in array lists. With the use of generics, the problem is simply eliminated as the required functionality is embedded within the program. The following is an example when casting does not need to be explicitly mentioned:

```
package java11.fundamentals.chapter13;

import java.util.ArrayList;
import java.util.List;

public class Casting {
    public static void main(String args[]) {
        List<String> mylist = new ArrayList<String>();
        mylist.add("I love Arraylist");
        String mystring = mylist.get(0);
        System.out.println(mystring);
    }
}
```

The above program produces the following output.

I love Arraylist

This use of generics ensures that there is no need to perform type casting and change items between primitive data types and the Wrapper classes in Java. Another key benefit of using generics is that it allows Java programmers to develop good habits

and then employ generic frameworks which are type safe, easier to read, and can be customized as and when required during a specific coding application. The main benefits of using generics is to have flexibility in your programs, while ensuring type safety of the objects and methods that you employ.

You can set up structures in your programs that offer improved usability as a complete package. Since you do not specify the fixed data types in your interfaces and generic classes, you can come up with a code that can work with different collections of datasets, such as ones that require a combination of integer and string values. You can avoid the problems that you will face when using non-generics and get a superior code, especially with the use of a single interface definition that offers a strong type control over the employed data objects.

### 13.2.2 Using Generics in Java

There are various generic elements that you can employ in Java. It is possible to appreciate the benefits of Java if you can compare it with the use of a traditional code to handle the same task. Let us take the example of the Box class for this purpose, where we will describe how it is possible to create a generic version of the class, ensuring that we get excellent benefits when we go ahead with programming and expanding on the created base class.

A simple class will employ private objects where we will have to use two methods of `set()` and `get()` to ensure that the methods employed can use the class object. This does not mean that we can control how the object will be employed during the compilation phase, as it is possible to place a String when the object type is set to work as an integer object in the program code implementation.

This problem is easily resolved when we create a generic version of the same class. A generic version is created using the angle brackets “`< >`”. It ensures that the type declaration that we use employs a variable that we can protect. The following is the code example that shows the use of a generic class, which will offer enhanced type safety:

```
public class Box<T> {
    // where T describes the Type of the object use
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Here, we can clearly observe that the use of the type “`T`” that we have set up in the generic class declaration replaces all instances of the use of an object in the code. This ensures that the type variable can follow any array, class, another type variable, or any other non-primitive value providing better control and enhanced functionality in the Box class.

There are some conventions that you must follow when employing generic programming. The type parameters are always single capital letters. As it is difficult to use any name without setting up conventions, you may find it difficult to understand generic programs created by other developers. The common letters used by programmers often employ specific parameters that denote the following values:

- 1. E:** It denotes an element and you will find it employed by the Java Collections Framework, which we will define in subsequent sections.
- 2. K:** It denotes key and will be employed for pointing to various data objects and displaying the object information accordingly.
- 3. N:** It stands for number value, where we aim to employ them as parameters for the generic code.
- 4. T:** It denotes type. We can employ a particular type that we want to use in the program and this will offer type safety, which is a characteristic feature of generics in Java.
- 5. V:** It describes value. We can employ the type accordingly when we implement this in a generic class or object.
- 6. S, U, etc.:** These denote that we are employing multiple types. The letters will be denoting them successively, such as S for 2nd and U for 3rd types.

Once you have set up a general class, you need to implement an invocation which needs to present a concrete value. This can include an instance such as that of integer, which is an object. You pass the type argument to your generic class, which is different from using an argument with a class method. This locks up the generics and ensures that it is not possible to employ wrong arguments. Remember, the code will not generate a new object, rather, it will employ the use of a parameter and will produce a reference to your original generic class, which will be of the Integer type in your example:

```
Box <Integer> intBox;
```

Once you have instantiated a generic type in Java, you can continue to work with it in the program, using different type interfaces as well as employ various parameters. We have already described that it is possible to have any number of type parameters in a generic class. This allows us to use objects that may belong to various Wrapper classes, as a common use is to implement functionality for both Integer and String objects.

One way to create a generic interface is to set up an ordered pair of parameters. This can then be set to have keys and values as parameters, allowing the use of the programs in a variety of ways. The use of a generic interface is easier as you do not have to worry as to whether you are using types or objects, as autoboxing in Java will always ensure that it is possible to pass data types to a class, which are automatically converted according to the required use.

It is also possible to use parametrized types that are present within the confines of a generic class definition. Creating such a type is possible in the following manner, if we continue with our earlier example:

```
Box<Integer> intBox = new Box<>();
```

Now, this creates a raw type. This use of code may not be generic in some older versions, but Java fully supports the use of API classes and other detailed generic functionality, which is included in the compiler functionality.

### 13.2.3 Generic Methods

Now that we have explained the use of generic classes, especially the ones that work with parametrized types, it is time that we describe the generic methods. Creating a generic method is similar to creating a generic type, but the advantage here is that the scope of this type remains limited to a method. It is possible to set up both static and non-static generic methods, while they can also be implemented in the form of class constructions.

If we take the example of a static generic method, we find that the type parameter must be defined before the return type of the method is mentioned in the code. An example for invoking a method for this purpose may be in the form:

```
Pair<String, Integer>pal = new Pair<>("grape", 3);
```

It is also possible to use type inference which allows the use of a generic method as a typical Java method. This means that there is no need to specify the type of parameters in the angle brackets.

Now, we describe the benefit of using a generic method, which is to bound the parameters and ensure only restricted types that can be employed as method arguments. This can be achieved by creating the upper bound for the type and use the “extends” keyword as shown in the following example, like `<U extends Number>`.

```
public class Box<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(15));
        integerBox.inspect("some text"); // error: is presented since this would still be
returning a String
    }
}
```

This is an excellent example of using a method which works with a locked number type. Since we have inserted some string in the inspect method, this directly translates to a String object. This means that the compiler finds that the method cannot work with the available information, since the parameter describes a number extension limiting the parameters that the integerBox can employ. It is also possible to use further complications such as multiple bounds, but we are limiting our use here to the examples that we have presented.

An important concept to understand is that Java is an object-oriented programming (OOP) language. An important principle of OOP is that of inheritance. Inheritance allows creating relationships between different objects and classes. The same case is possible when using generics, where we can set up relations between different elements of generic code.

Creating subtypes of generic classes and interfaces is possible by either extending the class or implementing the interface. The relationship that we create between the various types of classes is possible by using clauses to define it. Here is an example:

```
interface MyList<E,T> extends List<E>{
}
```

This is a simple example where `MyList` is a generic interface which extends the `List` with a single parameter value. Here, it is possible to see that a generic list capable of accepting a string object can have a subtype which can accept an `Integer` object as well as a `String` object. There are wildcards as well that can be employed when setting up generic code. “?” is the most common wildcard and it denotes a currently unknown type. It can be used for a local variable, a field, a parameter value, and a return type.

Generics in Java are designed to check for errors at the compile time. It does not have a use at the time of runtime. The compiler has the function of “type erasure” as an important feature, which eliminates all the type checking code from the final program when creating the byte code. It may cast any type if it is required. The parameterized types do not create any new classes, and therefore, generics in Java do not have any additional overhead runtime resource use.

There are some things that are locked in Generics, such as creating subtypes at a fundamental unit. It is also not possible to create Generic arrays as they will not compile. The idea behind using generics is to create and store specific objects in your Java program. Since the type erasure occurs before the execution phase, the parameters cannot be designed to set up during the runtime phase of the program. This may restrict the programming approach, although it is possible to set up higher parameters such as objects, which still allow the use of subunits such as `Integer` or `String`.

Java Generics are constantly on the rise and developed in its advanced forms. Java 10, which came out in March 2018, has the experimental Project Valhalla present in it. It contains several generics improvements, such as the use of specialized lists and reified generics, which ensure that the actual types can be made available at runtime, even when employing generic code.

Before we can explore the world of Collections, it is important to understand two most important methods that are needed for collections to work properly. These methods are `hashCode()`, and `equals()`. In the following section, we will see why these methods are important and why one should override them.

### 13.2.4 Overriding `toString()`, `hashCode()`, and `equals()`



Everything in Java other than primitives is an object. Anything you can think of like exception, events, or arrays extends from `java.lang.Object`. Hence, everything contains the methods shown in Table 13.1 that reside in `java.lang.Object`.

**Table 13.1** Methods from `java.lang.Object`

Method	Description
<code>boolean equals (Object obj)</code>	This method is used to check if two given objects are meaningfully equivalent
<code>void finalize()</code>	This method is used by the garbage collector when it determines that there are no more references to the object
<code>int hashCode()</code>	Hashtable, HashMap, and HashSet Collection classes use hashing; this method returns hashcode int value for an object
<code>final void notify()</code>	This method is used to wake up a thread which is waiting for the object's lock on which it is used
<code>final void notifyAll ()</code>	This method is used to wake up all the threads that are waiting for the object's lock on which it is used
<code>final void wait()</code>	This method is used to make current thread to wait till other thread class <code>notify()</code> or <code>notifyAll()</code>
<code>String toString()</code>	This returns the text representation of an object

### 13.2.4.1 Overriding `toString()` Method

This method is useful in giving some meaningful information about the object. If you do not override this method, the object's default `toString()` will be called, which does not give any meaningful information. It only returns the classname followed by @ symbol and the object's hashcode. Let us see the following example:

```
package java11.fundamentals.chapter13;
public class Car {
    public static void main(String args[]) {
        Car car = new Car();
        System.out.println(car.toString());
    }
}
```

The above code gives the following result:

**java11.fundamentals.chapter13.Car@6e8dacdf**

Similarly, let us see another example of `toString()` in a side-by-side comparison view.

Without <code>toString()</code>	With <code>toString()</code>
<pre>class Student{     int rollno;     String name;     String city;      Student(int rollno, String name, String city){         this.rollno=rollno;         this.name=name;         this.city=city;     }      public static void main(String args[]){         Student s1=new Student(101,"Raj","lucknow");         Student s2=new Student(102,"Vijay","ghaziabad");          System.out.println(s1); //compiler writes here s1.toString()         System.out.println(s2); //compiler writes here s2.toString()     } }  Output:Student@1fee6fc Student@1eed786</pre>	<pre>class Student{     int rollno;     String name;     String city;      Student(int rollno, String name, String city){         this.rollno=rollno;         this.name=name;         this.city=city;     }      public String toString(){//overriding the toString() method         return rollno+" "+name+" "+city;     }      public static void main(String args[]){         Student s1=new Student(101,"Raj","lucknow");         Student s2=new Student(102,"Vijay", "ghaziabad");          System.out.println(s1); //compiler writes here s1.toString()         System.out.println(s2); //compiler writes here s2.toString()     } }  Output:101 Raj lucknow 102 Vijay ghaziabad</pre>

In the overridden method, you can give some meaningful information about the object, which describes the object for better understanding. For example, if you have a customer class, you can give customer related information, such as name and account number ,so when someone calls `toString()` on that object, it will give useful information about that customer.

```

package java11.fundamentals.chapter13;
public class ToStringExampleWithOverriddenToString {
    public static void main(String args[]) {
        ToStringExampleWithOverriddenToString tse = new ToStringExampleWithOverriddenTo-
String();
        System.out.println(tse.toString());
    }
    public String toString() {
        return "This is an overridden method";
    }
}

```

The above code gives the following result.

**This is an overridden method**

As you can see, it prints the text which we have provided in the `toString()` method.

### 13.2.4.2 Overriding `equals()` Method

As you have seen, a simple way to compare two variables is to use `==`. However, `==` is going to check if two references are pointing to the same object or not because `==` simply looks at the bits in the variable, and checks if they are equal or not. If you only care about checking if two object references are equal, feel free to use `==`. However, this is not the case most of the times.

In case of objects, we always want to make sure they are meaningfully equal. Now, `==` can only tell us if these two references pointing to the same object on the heap or not. Hence, we need to override `equals()` method and compare the necessary attributes of the class, which will tell us if they are truly equals. It is also important to make sure which variables you use for the comparisons. If you do not use unique ones per object, you are going to get the wrong result. For example, in case of a Car class, two objects of Car may have similar attributes, such as color, make, and model. If you are using these attributes in the `equals()` method to compare, you may end up concluding that two completely different cars are same, even though they are owned by different persons. Hence, in this case, you may want to compare on a unique element of that class, such as registration number. This will give you the correct result for the comparison.

#### 13.2.4.2.1 Importance of Overriding `equals()` Method

For collections, it is important to override class's `equals()` method because without overriding and giving meaningful equal comparison, it is not possible to find the object in the collections. Hence, we will not be able to use the object of this class as a key in a hashtable or HashMap. Let us explore this in detail. If we do not override `equals()` method, then the Object's `equals()` method will be used instead as everything extends from Object. This Object's `equals()` method uses only `==` operator for comparisons. This `==` comparison means two objects are treated equal if those two references refer to the same object.

Now, let us look at an example where we will not be overriding the `equals()` method. Let us assume that we have two houses that are being sold at the government registry office. Your program is to put house object and owner object in a HashMap, where the house object is the key and owner is the value. Now, when we want to retrieve the information after finishing the process, we need to provide the house object to get the owner's information. This is tricky as we now need to provide the exact reference to the object we used as the key when we added it to the collection. Remember, we cannot create an identical house object again and use it for search. It is because this newly created object will refer to a different location on the Heap, and since Object's `equals()` method is using `==` to compare two objects, it is only comparing the objects' memory locations to check if they are same. Even though these two objects are logically equal as per the `==` comparison, they are different. Hence, we will never be able to create identical object and we will fail to get the data out of HashMap. This is why we will not be able to use an object as a hashtable key or HashMap key without overriding `equals()` method and comparing two objects for logical equivalence.

How do we solve this problem with overriding `equals()` method? We need to make sure we use the suitable attribute to compare two objects so that they can be meaningfully equivalent despite referring to two different objects on the Heap. So, in

our example, let us assume that we can use house address as a comparison attribute in the `equals()` method. Now, we can compare two addresses to make sure these two houses objects are same and hence we can retrieve the owner object from the `HashMap`. However, we can still improve on this a little. Instead of using house object as the key, we can simply use house address as the key so that we will not need to create an identical object reference in the future to retrieve the owner object. We can simply use address String to get the owner object from the `HashMap`. This is possible as String and Wrapper classes work well as keys in hashtables, `HashMaps`, etc., as they override the `equals()` method.

```
package java11.fundamentals.chapter13;
public class EqualsMethodTest {
    public static void main(String[] args) {
        Car myCar = new Car("Mercedez Benz", "S Class", "MH05 12345");
        Car carInGarage = new Car("Mercedez Benz", "S Class", "MH05 12345");
        if (myCar.equals(carInGarage)) {
            System.out.println("Yay!!! This is my Car!");
        }
    }
}
class Car {
    private String brand;
    private String model;
    private String registrationNumber;
    Car(String brand, String model, String registrationNumber) {
        this.brand = brand;
        this.model = model;
        this.registrationNumber = registrationNumber;
    }
    public boolean equals(Object o) {
        if (o instanceof Car) {
            Car car = (Car) o;
            if (car.getBrand() == this.brand && car.getModel() == this.getModel() && car.getRegistrationNumber() == this.getRegistrationNumber()) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    }
    public String getBrand() {
        return brand;
    }
    public void setBrand(String brand) {
        this.brand = brand;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
    public String getRegistrationNumber() {
        return registrationNumber;
    }
    public void setRegistrationNumber(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }
}
```

The above code produces the following result.

Yay!!! This is my Car!

In the above example, class Car overrides `equals()` method which compares car brands, models and registration numbers using `==` operator. Since we are comparing Strings, we can safely use `==` operator for comparison. String is immutable which means string's content will not change once created. And any string objects that we create are stored in the String Constant Pool. This pool cannot have two objects with same content. Hence, if we create another object with the same content, it will only point to the same object in the heap instead of creating a new reference. Therefore, `==` on two strings will give proper result.

#### 13.2.4.2.2 The `equals()` Method Contract

Every class overriding `equals()` method must adhere to the `equals()` contract mentioned below, which is taken from the Java Docs:

1. **Reflexive:** For any reference value `x`, `x.equals(x)` should return true.
2. **Symmetric:** For any reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
3. **Transitive:** For any reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` must return true.
4. **Consistent:** For any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
5. For any non-null reference value `x`, `x.equals(null)` should return false.

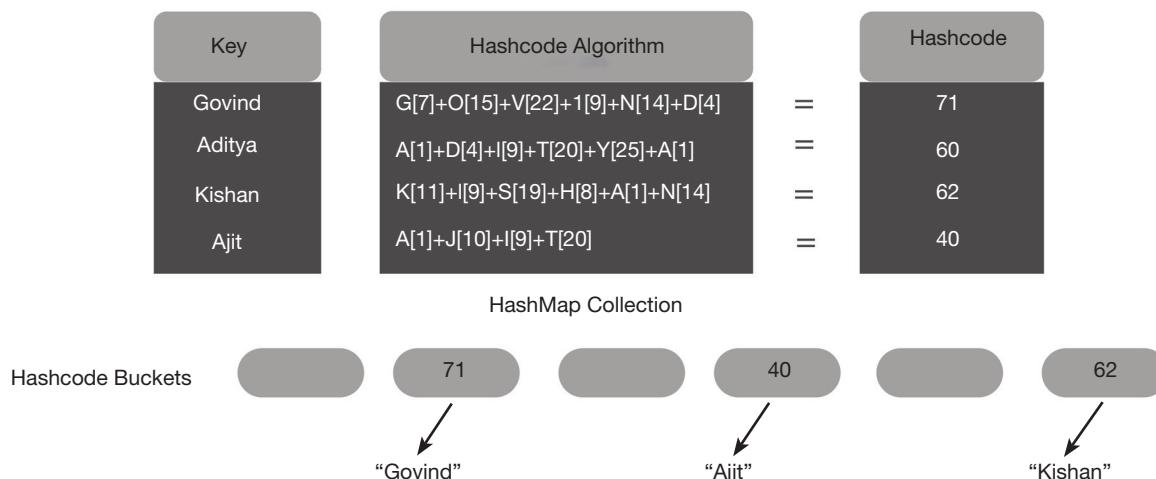
Next, we will look into `hashCode()` method, as `equals()` and `hashCode()` are bound together by a joint contract. This contract states, “if two objects are considered equal using the `equals()` method, they must have identical hashcode values”.

Hence, we need to look into overriding `hashCode()` as well to make sure we create truly meaningful equal objects.

#### 13.2.4.3 Overriding `hashCode()`

In collections, you can think of hashcode as an object ID but not necessarily unique, which is used to increase the performance of large collections of data. Collections like `HashMap` and `HashSet` use the hashcode of an object to see how to store it in the collection and for its retrieval later. Hence, hashcode plays an important role in collections. We need to make sure we have as unique a hashcode as possible to increase the performance of the collections.

Let us try to understand this with the help of an example. In a room, there are 1000 buckets placed, with labels from 1 to 1000. You are given a bunch of slips. Each slip has a name on it and you need to determine a numeric value of each name by adding the letters' numeric equivalents like A=1, B=2, C=3, etc. Each name will give you a number. You need to place this name slip in the bucket that has this number label on it.



Now, if someone comes with a name and asks to get a slip for that name, we can quickly get the number for corresponding to that name and look for the slip in the bucket. This is how collections work. While creating this hashCode generator algorithm, we need to make sure we generate the same hashCode for the same input. So, each time we pass the same name we get the same number back.

As you may have noticed, we may have more than one number for many names that share same letters, such as Ayaan and Nayaan. Since they share the same letters, they will have same number if we add all the characters. It means one bucket may have multiple slips in it. This is acceptable however, if we make our algorithm efficient and we have distributed load in the buckets. This will also speed up the search operation.

Search is a two-step process. First finding the right bucket using `hashCode()` and then searching the bucket for the right element using `equals()` method.

Now, consider the case where our hashCode generator algorithm does not give us consistent result, and we may get different hashcodes at different times by passing the same name. This could be a problem, as when we first added a name in these hash collections, the name would have gone to a bucket with the name's hashCode value. Now, when we try to retrieve the same name, this hashCode generator algorithm gives different hashCode even though the name is same. In this case, we are not going to find the bucket as our name, when we first added is in completely different bucket than we are looking into. Because of this hashCode generator's inconsistent behavior, we will find no-match even though the objects are equal. Hence, it is important to note that for two objects to consider equal, their hashCode must also be equal. This is because if the hashCodes were different, the equal test would fail due to scanning into a wrong bucket.

Let us look at the following example, which shows how to override `hashCode()` method.

```
package java11.fundamentals.chapter13;
public class HashCodeTest {
    public int myVar;
    HashCodeTest(int myVar) {
        this.myVar = myVar;
    }
    public boolean equals(Object o) {
        //We need to check if the object is instanceof of HashCodeTest class. If not we
        can safely return false
        if(o instanceof HashCodeTest) {
            HashCodeTest hTest = (HashCodeTest) o;
            if (hTest.myVar == this.myVar) {
                return true;
            } else {
                return false;
            }
        }else {
            return false;
        }
    }
    public int hashCode() {
        return (myVar * 23); //You can use your own logic. We tried to multiply by a
        prime number.
    }
}
```

#### 13.2.4.3.1 The `hashCode()` Contract

The following contract is taken from the Java API documentation as is. This describes the `hashCode` contract:

1. Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer, provided no information used in `equals()` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
2. If two objects are equal according to the `equals(object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.

3. It is NOT required that if two objects are unequal according to the `equals(Java.lang.Object)` method, then calling the `hashCode()` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

Now, let us try to interpret this contract (Table 13.2).

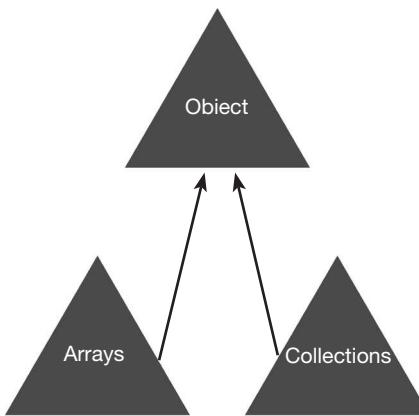
**Table 13.2** `hashCode()` contract

Condition	Required	Allowed but Not Required
<code>x.equals(y) == true</code>	<code>x.hashCode() == y.hashCode()</code>	
<code>x.hashCode() == y.hashCode()</code>		<code>x.equals(y) == true</code>
<code>x.equals(y) == false</code>		No <code>hashCode()</code> requirements
<code>x.hashCode() != y.hashCode()</code>	<code>x.equals(y) == false</code>	

## 13.3 | Collections



Object collections are important in Java. They are available as a framework and provide the architecture required to store a collection of objects. The framework and the supporting APIs allow programmers to not only store the collections but carry out various information manipulations. This ensures that it is possible to perform functions, such as String changes and modifications. Figure 13.1 shows Arrays and Collections relationship with Object.



**Figure 13.1** Arrays and Collections extends Object.

The use of collections is always optional and allows you to employ the set of objects as a single coherent structure. Usually, the framework contains interfaces and classes to set up the object groups and provide the required functionality to the programmers. This functionality is accessible through the use of collection APIs.

A key point that arises is that arrays also hold primitive objects in Java and therefore, can also be considered as object collections. They are also managed as a singular group item. Thus, it is possible to sort, modify, and work on the data items that are present in an array, as it effectively works as a pointer to the data objects. However, collections are different because their capacity is not assigned when they are instantiated by a programmer.

Collections do not hold the elements of the primitive types. Rather, they hold complete objects belonging to the Primitive Wrapper classes, including Long, Integer, and Double. The earliest Java platform did not have collections implementations, and object manipulation operations had to be carried out using various array options. These were difficult to control and would produce problems during interfacing to have standardized members.

### 13.3.1 Collections in Java

A concurrency package was developed, which could handle the Java collections. The `java.util.Collection` contains all the collections implementation facilities that we now have available in JDK and IDE software tools for program development. Java architecture combines collections with program generics and offers three types of collections.

1. The first type is the use of ordered lists. These are important when you want to insert objects in a specific order, such as creating a waiting list.
2. The second type is in the form of a dictionary/map that provides a set of information, which can be searched using a reference key to obtain the value of a particular object.
3. The third type is a set. Sets are simply unordered collections. This means that you cannot have similar objects because there is no way to have distinct objects without having an order.

Each of these collection elements has its specific advantages. The ordered lists are excellent as a more flexible way to implement an array. They provide a storage space for their elements that have a better order. Duplication is possible as long as each element is properly placed in its specific position. All positions are available for search, and the listing can be done by either using linking or creating them in the form of an array styled listing.

Another option is the creation of stacks. The stack library in Java allows the use of creating a stack of objects. The stack works on the Last In, First Out (LIFO) principle, when returning objects. This means that the latest item present on the stack will be the first one to be returned, therefore, earning the name of a stack, where you can take out the item on the top first.

There are five operations present that ensure that any vector can be created as a stack. They are as follows:

1. The first method `push()` is used to place any new object in the stack.
2. The second method `pop()` is used to remove an object from the stack.
3. The third method is to look at the top item of the stack.
4. The fourth method finds whether a stack is empty or not.
5. The final method finds a specific item in the stack and describes its positional distance from the top of the stack.

Remember, all stacks are empty when they are created in a Java implementation.

There is also an interface that describes a queue data structure. This is a collection where the items are ordered, but the order remains the one in which they are stored in the stack. All additions are added to the end of the object list, while all removals happen from the front. This is termed as *linked list implementation*. This presents the First In, First Out (FIFO) scheme, which provides a different implementation method, separating it from the LIFO option of the first type of stack.

So, to summarize what we have discussed so far, the following are a few basic operations needed to perform with collections:

1. Adding new objects to the collection.
2. Removing objects from the collection.
3. Looking for objects or group of objects in the collection.
4. Getting an object from the collection without removing it.
5. Looking for a specific object at a specific index.
6. Iterating through the collection one object at a time.

### 13.3.2 Benefits of Java Collections

The Java Collections Framework offers excellent benefits which are as follows:

1. The main advantage it offers is that it reduces the total programming effort that a developer must go through when creating long and complex programs. This is possible by setting up customized data structures and algorithms like custom collections that can be used as a plumbing system to deliver resources throughout the program.
2. It is important to create a strong data object system in programs. Whether you want to perform integration between various program elements or build operational capability between the different Java APIs, Java Collections provide you the opportunity to create adapter objects. In fact, it is also possible to come up with a conversion code, which allows a Java program to use different APIs and employ them to the optimal advantage of creating efficient and cost-effective programs.
3. The execution speed and the quality of the programming code are important parameters, especially for modern cloud applications. It is possible to increase the program efficiency by using the Collections Framework.

4. When you set up useful data structures in your program, you can come up with multiple interfaces. They are interchangeable because of the use of specific data collections. You have set up data structures that allow you to completely devote yourself to creating better code, while not worrying about how to tackle your objects and data storage elements. The collections allow you to create efficient data structures that can deal with all the information. They are also more flexible than simple arrays that are available in other various programming languages.
5. A key benefit is that unrelated APIs and programming elements can easily work together by using the strong collection interfaces. APIs can interact with each other and share information in the form of the collections that are empowered through the Java Collections Framework. You can employ node names and come up with a strong implementation solution, which is beneficial in a variety of environments.
6. Another advantage is that the use of collections available in Java make it easier to use new APIs and other tools. Naturally, collections can be used to pass data to the APIs as well as obtain the required inputs. This allows programmers to only have the knowledge of collections and then use any API that they require in a particular program. You do not have to learn from scratch every time there is a need to use a new API for a specific Java functionality. In fact, expert Java programmers create their own APIs to help other developers with code simplification and provide support for functions that may be required repetitively in a number of situations. There is no need to reinvent the wheel in terms of sharing objects, data, and information when providing operational capabilities in new APIs. All you need to do is make them compatible with the Java Collections Framework for the required level of connectivity and support.
7. The new data structures that are prepared in your programs can be made to conform to the collection interfaces in Java. This creates a database of objects and information that you can employ later as well. New algorithms can be easily made available for reuse in their implementation if they are prepared according to the needs of the collection interfaces compatible in Java.

### 13.3.3 Collection Interfaces

Java Collections interfaces form the foundation of the framework. They are often employed with the use of generic coding, which is then combined with a specific data collection. The interfaces that are created are perfect for reducing the runtime errors as all objects are checked during compilation. Java does not set up separate interfaces to ensure that the core collection is easily managed. It throws an exception if an unsupported operation is implemented using the Collections.

The interface here sets up the root of the collection hierarchy. There is a group of objects that make up the collection. The interface allows developers to employ various methods which allow them to check the elements in the collection, as well as perform the functions of removing, adding, and performing an iteration on the available collection elements.

Table 13.3 gives the list of core collection interfaces.

**Table 13.3** List of core collection interfaces

Collection	Set	SortedSet
List	Map	SortedMap
Queues	ConcurrentMap	NavigableMap
ConcurrentNavigableMap		

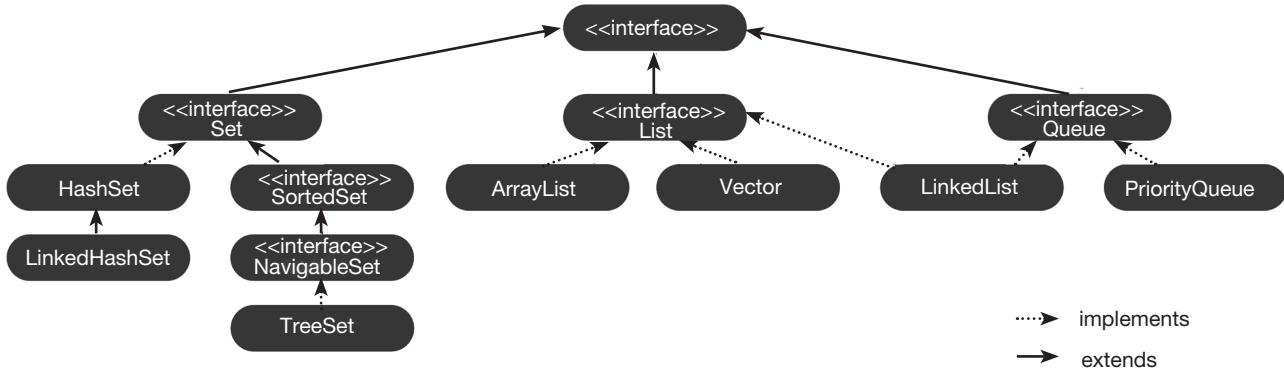
Collections Framework has many collections but not all of them implement the Collection interface. For example, Map interface does not extend Collection interface or none of the classes implement the Collection interface.

The word Collection is little confusing in Java as there are three overloaded uses of the word “collection”. Therefore, you should pay close attention to the word and make sure you refer it properly.

1. **Collection (note lowercase c):** This represents any of the data structures in which objects are stored and iterated over.
2. **Collection (note capital C):** This is the `java.util.Collection` interface. This interface is the one which is extended by List, Set, Queue, etc. Also, note that there is no direct implantation of Collection interface. Only other interfaces extend this interface.
3. **Collections (note “s” at the end and capital C):** This is not an interface but a class from the `util package.java.util`. Collections class. It contains static utility methods to use with collections. For example, `sort`, `emptyList`, `emptySet`, `emptyMap`, `addAll`, `binarySearch`, etc.

In this chapter, we will mainly talk about Collection interface. The following are different aspects of Collection interface:

1. **List:** This Collection interface represents list of things.
2. **Set:** This Collection interface represents unique things.
3. **Map:** This Collection interface represents things with a unique ID.
4. **Queue:** This Collection interface represents things arranged by the order in which they are to be processed.



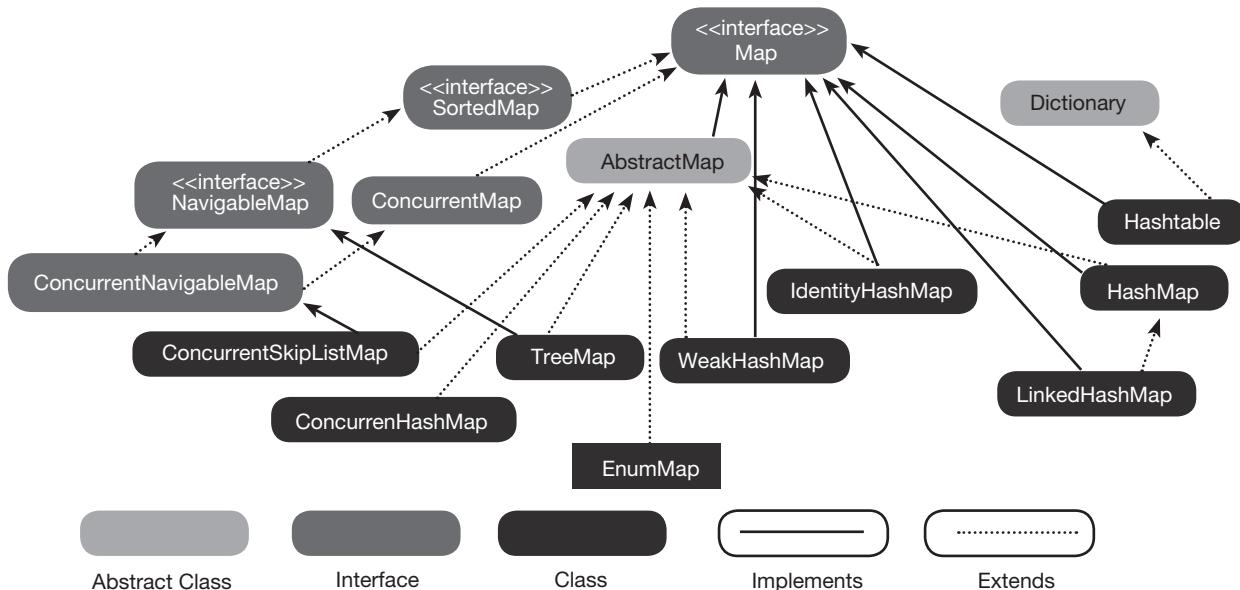
**Figure 13.2** Collection Interface.

Figures 13.2 and 13.3 give us a good idea about some of the core interfaces and implantation classes. In Section 13.4, we will explore these classes in detail. The following are the subsets of these interfaces.

1. Sorted
2. Unsorted
3. Ordered
4. Unordered

The implemented classes can be one the following:

1. Unsorted and Unordered
2. Ordered but Unsorted
3. Ordered and Sorted



**Figure 13.3** Map Interface.

It is important to note that an implemented class cannot be sorted but unordered. As you may have figured, this is because sorting itself is a kind of ordering. Now, let us take the example of HashSet, which is an unordered and unsorted set. On the other hand, the LinkedHashSet is ordered but not sorted. The order of LinkedHashSet is maintained on an insertion basis. The order is set as you add elements to it.

In order for us to understand the concept of collections, first we need to understand the concept of iteration. For a simple “for loop”, we access elements in order of index such as 0, 1, 2, 3, etc. However, iterating over a collection is quite different. Elements are get collected one by one; so there is an order as it starts from the first element. However, in some cases, the first position is not defined the way we think. Take the example of Hashtable, in which there is no first, second, or third order. The elements are simply placed in a chaotic order based on the hashCode of the key. Hence, the first element is determined at the time of iteration because it constantly changes as the collection changes.

Let us explore Ordered and Sorted concepts.

- 1. Ordered collection:** The collection is said to be ordered when iteration can happen in a specific order. As we have seen, Hashtable is not ordered since we cannot determine the order of the elements. The order is determined by the hashCode of the elements and constantly changes as the collection changes. On the other hand, ArrayList is ordered as it inserts the elements on specific indexes. You can determine which index to use to add an element to ArrayList. Hence, it is ordered via index position. However, LinkedHashSet keeps the insertion order; that means that the first element inserted will be on the first order and last element inserted will be on the last order. Some collections use the natural order; that means that they are not only ordered but sorted (by natural order, such as A, B, C, ... and 1, 2, 3, ...). Now, natural order is very straightforward in case of alphabets and numbers, but what about objects? Objects do not have a natural order. So how do we determine the objects’ natural order? For this, Java provides an interface called *Comparable*. This interface has a provision to define a natural order. It means, developers have the freedom to define the natural order for the objects. This rule can be like deciding order based on one of the instance variables, for example, price, age, name, etc. Java also provides another interface called *Comparator*, which can be used to define sort order for objects.
- 2. Sorted collection:** This type of collection is ordered based on some rule(s). These rules are called *sort order*. This sort order is not determined by the time the object gets inserted into the collection, or accessed the last time, or on which position it was inserted. This sorting is based on the properties of the objects that are added to the collection. It means that we simply insert the object into the collection and let the collection determine the order of that element based on the sort order. A collection that keeps an order like ArrayList are not really sorted unless they are explicitly sorted based on some type of sort order. In most cases, the sort order is the natural order of that object.

### 13.3.4 Collections Classes

Java provides amazing classes that you can employ to carry out the required functions in your program. The main class is the HashSet class. It is backed by the API of HashMap. It does not provide an iteration order and is capable of implementing bucket lists. The initial capacity can be set up in this class, ensuring that it follows a fixed set of capacity increments that save the available program resources. It provides a constant time performance where the elements are dispersed properly in the available buckets.

The TreeSet Class makes use of a TreeMap. The natural order to the creation time is often employed when constructing this Collections class. This class is employed when we want to maintain a consistent structure. It employs comparisons to check different elements, ensuring that it offers the use that we are looking for in a class. Table 13.4 lists some of the top options that are available in Java for use in Collections.

**Table 13.4** List of Collections Classes

Map	Set	List	Queue	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				

Table 13.5 shows all the classes and indicates whether they are ordered and/or sorted.

**Table 13.5** Properties of Collection Classes

Class	Implements	Ordered	Sorted
HashMap	Map	No	No
HashTable	Map	No	No
TreeMap	Map	Sorted	Sorted by natural order. It can also be sorted by custom comparison rules.
LinkedHashMap	Map	Ordered by insertion order. It can also be ordered by last access order.	No
HashSet	Set	No	No
TreeSet	Set	Sorted	Sorted by natural order. It can also be sorted by custom comparison rules.
LinkedHashSet	Set	Ordered by insertion order	No
ArrayList	List	Ordered by index	No
Vector	List	Ordered by index	No
LinkedList	List	Ordered by index	No
PriorityQueue	Queue	Sorted	Sorted by to-do order

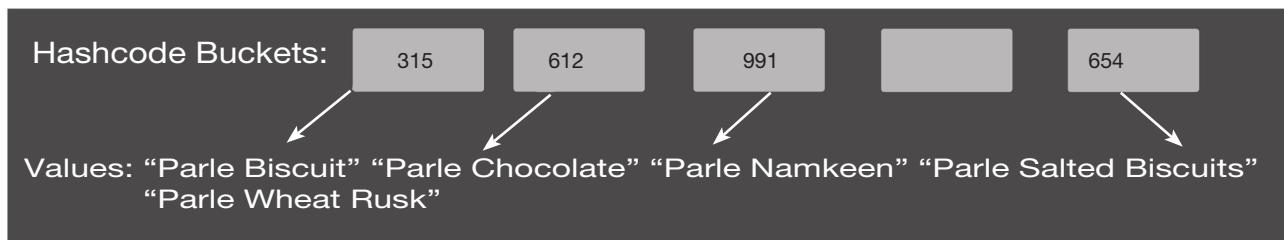
## 13.4 | Implementing Collection Classes

There are several ways to implement Java Collections functionality. Here, we will explain each concept to build up the desired level of knowledge required for using Collections with Generics. We will describe various API settings that allow the use of collection functionality and provide a detailed discussion on how these concepts are implemented in the language.

### 13.4.1 Map Interface

Map is a key value pair collection. It takes unique identifier as a key that acts like an ID. Both key and value take object as input. This allows us to search for a value based on the key. Maps and Sets both rely on the `equals()` method to check if the two keys are the same or different.

Figure 13.4 shows how values are stored as key–value pair in Map. Hashcode buckets contain the keys and point to its corresponding values. In the following example, key 315 is associated with value “Parle Biscuit”, 612 is associated with “Parle Chocolate”, etc.

**Figure 13.4** Key–value structure in Map.

#### 13.4.1.1 HashMap

HashMap is a collection class that uses the system of pairs, where one is the key and the other is the corresponding value. The syntax is in the form of `HashMap<K, V>`, where the key comes first while it is followed up by the corresponding value. The objects that are stored in this collection do not have to be ordered as it is employed to find any value by using the corresponding key. It is possible to set up null values in this collection class.

This class is imported from `java.util.HashMap`, which describes its path in the utility library. The following is an example of using this class, in which we have removed the importing of the various other requirements for the program:

```
package java11.fundamentals.chapter13;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<Integer, String> hmap1 = new HashMap<Integer, String>();
        hmap1.put(14, "George");
        hmap1.put(33, "Paul");
        hmap1.put(16, "Jane");
        hmap1.put(7, "Brian");
        hmap1.put(19, "Jack");
        Set set1 = hmap1.entrySet();
        Iterator iterator1 = set1.iterator();
        while (iterator1.hasNext()) {
            Map.Entry ment1 = (Map.Entry) iterator1.next();
            System.out.println("The value is: " + ment1.getValue() + " and key is: " +
ment1.getKey());
        }
        String va = hmap1.get(2);
        System.out.println("Index 2 has value of " + va);
        hmap1.remove(16);
        Set set2 = hmap1.entrySet();
        Iterator iterator2 = set2.iterator();
        while (iterator2.hasNext()) {
            Map.Entry ment2 = (Map.Entry) iterator2.next();
            System.out.println("Now value is " + ment2.getValue() + "and key is: " +
ment2.getKey());
        }
    }
}
```

The above program produces the following output.

The value is: Jane and key is: 16
The value is: Paul and key is: 33
The value is: Jack and key is: 19
The value is: Brian and key is: 7
The value is: George and key is: 14
Index 2 has value of null
Now value is Pauland key is: 33
Now value is Jackand key is: 19
Now value is Brianand key is: 7
Now value is Georgeand key is: 14

This is a detailed program that shows the complete use of the `HashMap` class. All properties are perfectly shown by the use of the example. The first section sets up a `HashMap` collection and places five values with five keys in it. They are displayed using the while conditional loop and follow an alphabetical pattern according to the values, which is possible with the use of the iterator. Next, we remove the value associated with the key of 16 which will remove the "Jane" entry. Now, when we perform

the same operation, we will find the list printed only four values. This time the “Jane” value and its key is not present in the collection.



Can you insert integer as a key and image as value in HashMap?

There are some other excellent methods. The `clear()` method can clear a Map collection completely, while the `clone()` method creates a clone of one map to another, which can then be manipulated while ensuring that we also keep a legacy copy of the original collection. These represent some basic use of this class, allowing us to experiment with this type of collection, where the purpose is to create an unordered pair of object values.

### 13.4.1.2 Hashtable

Another collection class which is commonly implemented is that of the Hashtable. It is assigned from the `java.util.Hashtable` importing tree. It offers a slightly different implementation than the map, as it creates a table of keys and values, resulting in the production of synchronized set of objects, just like a truth table or a cartesian coordinate system.

It is possible to initiate it using the default constructor or to set up its size. It is also possible to read the elements of a Map in a Hashtable. Considering the earlier example in the HashMap given in Section 13.4.1.1, here is a program that describes the use of a Hashtable in Java:

```
package java11.fundamentals.chapter13;

import java.util.Enumeration;
import java.util.Hashtable;

public class HashtableExample {
    public static void main(String args[]) {
        Enumeration nms;
        String keys;
        Hashtable<String, String> hashtable = new Hashtable<String, String>();
        hashtable.put("Key1", "Adam");
        hashtable.put("Key2", "Brian");
        hashtable.put("Key3", "Charles");
        hashtable.put("Key4", "Dean");
        hashtable.put("Key5", "Peter");
        nms = hashtable.keys();
        while (nms.hasMoreElements()) {
            keys = (String) nms.nextElement();
            System.out.println("Key is " + keys + " & value is " + hashtable.get(keys));
        }
    }
}
```

The above program produces the following output.

Key is Key4 & value is Dean Key is Key3 & value is Charles Key is Key2 & value is Brian Key is Key1 & value is Adam Key is Key5 & value is Peter
--

In the above output, you can see the keys are printed in descending order, from Key4 to Key1, with the corresponding values. The last key entered is always presented at the end. There are various methods that are available for use in this collection class. It is possible to create an enumeration of the keys present in the table, while the `rehash()` method can enhance the size of the table and rehash all its keys. Both keys and the values can be searched within the collection, and it is also possible to get the return of the elements that are present as values in the table.

This class is excellent for use in a variety of conditions. It is perfect when we need to perform collection comparisons to find out something has changed. The table objects can be directly printed by giving the name of the hashtable identifier directly in the print statement.

### 13.4.1.3 TreeMap

`TreeMap` is a class which implements a navigable map in Java. It employs the natural ordering of the keys that are used to enter values. It is different from the `HashMap` since it creates a map according to the ascending order of its key values. However, this means that this class is not thread-safe and therefore, cannot be properly used with parallel programming in Java, unless it is kept safe.

Here, we present an example to show this behavior as compared to the one that we presented in `HashMap`. This will allow you to understand the use of this collection class, which is perfect in several scenarios where the specific order is important.

```
package java11.fundamentals.chapter13;

import java.util.Set;
import java.util.Iterator;
import java.util.Map;
import java.util.TreeMap;

public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<Integer, String> trmap = new TreeMap<Integer, String>();
        trmap.put(1, "Object 1");
        trmap.put(17, "Object 2");
        trmap.put(50, "Object 3");
        trmap.put(7, "Object 4");
        trmap.put(3, "Object 5");
        Set set = trmap.entrySet();
        Iterator iterator1 = set.iterator();
        while (iterator1.hasNext()) {
            Map.Entry ment = (Map.Entry) iterator1.next();
            System.out.print("key is: " + ment.getKey() + " and Value is: ");
            System.out.println(ment.getValue());
        }
    }
}
```

The above program produces the following output.

key is: 1 and Value is: Object 1
key is: 3 and Value is: Object 5
key is: 7 and Value is: Object 4
key is: 17 and Value is: Object 2
key is: 50 and Value is: Object 3

This program principles out objects not based on their value and setting. They are printed out with the help of the order of the key values. This means that the TreeMap which is created for the collection automatically comes up with the key order of 1, 3, 7, 17, and 50, which will correspond to their specific values and will be printed in that order.

#### 13.4.1.4 LinkedHashMap

The next implementation class is the LinkedHashMap. It can be presented as a combination of the facilities that are present in a hash table and then combined with the map interface that allows for creating a predictable iteration order for use in Java programs. It is different from a normal HashMap, because it uses a double linked list which connects to the entries that are present in this collections class. The linking defines the way iteration will be performed on the keys that are inserted into the map.

The added functionality provides a situation where we need access the insertion order of the values that become part of the collection. Here is an example to help you understand the use of this Java Collections class:

```
package java11.fundamentals.chapter13;

import java.util.LinkedHashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class LinkedHashMapExample {
    public static void main(String[] args) {
        // Declaring a HashMap
        LinkedHashMap<Integer, String> lihamap = new LinkedHashMap<Integer, String>();
        // Adding the elements to this collection map
        lihamap.put(21, "Abe");
        lihamap.put(35, "Drown");
        lihamap.put(1, "Jack");
        lihamap.put(3, "Karen");
        lihamap.put(100, "Lin");

        // Generating the required set
        Set set1 = lihamap.entrySet();

        // Displaying elements from this collection map
        Iterator iter1 = set1.iterator();
        while (iter1.hasNext()) {
            Map.Entry me = (Map.Entry) iter1.next();
            System.out.print("The key is: " + me.getKey() + " and Value is: " +
                me.getValue() + "\n");
        }
    }
}
```

The above program produces the following output.

The key is: 21 and Value is: Abe The key is: 35 and Value is: Drown The key is: 1 and Value is: Jack The key is: 3 and Value is: Karen The key is: 100 and Value is: Lin
--

The output of this program is different from previous examples. It will return and print the entire list in the order of the way the values are inserted in the program. The first value printed will be Abe with the last value printed will be Lin, in the same order. It is possible now to create chronological lists that you can use for the required functionality in your program. Remember, this map interface requires unique keys and it maintains the insertion order.

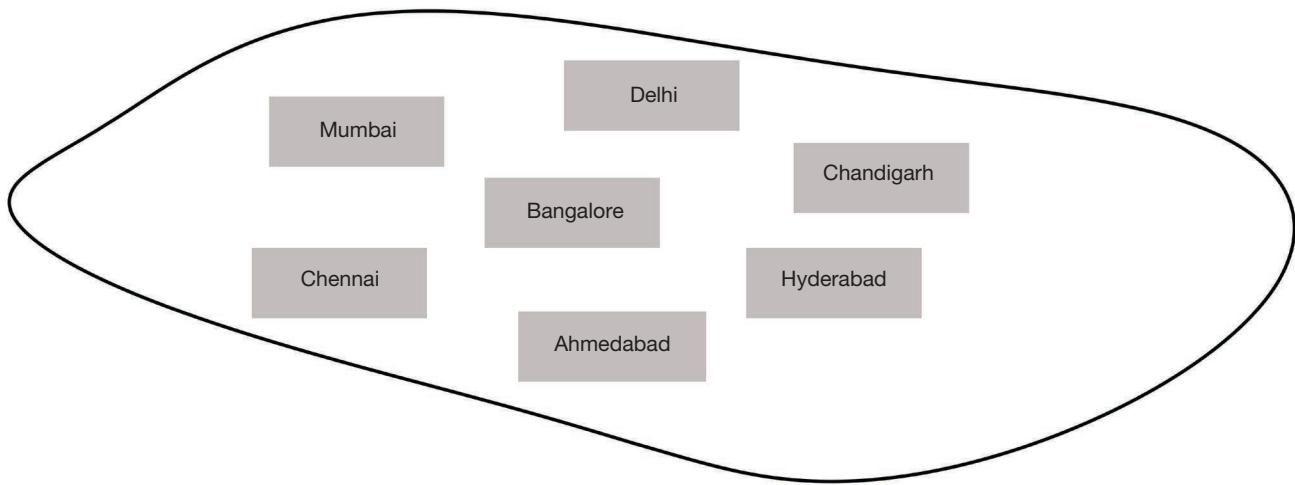
There are obvious benefits of using LinkedHashMap. Think of a situation where one module reads a map and copies it, and then produces results that completely depend on the order of the map elements it received.

This map provides access to all the basic operations on collection objects. It may have slower performance than a simple HashMap due to managing the order of the collections. However, performing an iteration is faster since it is only based on the size of the map, as we need access only to the order of the total entries.

Another important element to note is that the LinkedHashMap is not synchronization safe, since a thread can modify the map and affect the order of the stored collections. The knowledge of this type of map is important since it allows programming using Java to make informed decisions and pick the best options from the available set of services.

### 13.4.2 Set Interface

Set does not allow duplicates. Hence, each element is unique in a set. Similar to Map, Set also uses `equals()` method to check if two objects are equal. In case two identical objects found, only one of them will get inserted into the Set. Let us explore the concept of Set implementation. Figure 13.5 shows the representation of Set.



**Figure 13.5** Representation of Set.

#### 13.4.2.1 HashSet

Now that we have gone through the classes that employ the map interface, we will discuss the set-based classes that can be implemented from the Set interface in Java. This class does not follow any order; this means that the elements are returned in a random order. It also does not allow the use of duplicate values, as inputting the same value will simply replace the older one.

This set collection also accepts null values, but there can only be one null value stored in a HashSet. The iterator is fail-safe and therefore, it is not possible to modify the set once the iterator has worked on it. However, the remove method within the iterator can remove any value from the hash set without a problem.

The best way to understand the use of a HashSet is to go through a simple example. Here, we describe how programmers can use it as well as display the different items stored in the collection, which are easier to do when compared with the maps that require handling of the keys as well.

```

package java11.fundamentals.chapter13;
import java.util.HashSet;
public class HashSetExample {
    public static void main(String[] args) {
        // Declaring a HashSet
        HashSet<String> haset = new HashSet<String>();
        // Adding different elements including null ones
        haset.add("Apricot");
        haset.add("Mango");
        haset.add("Orange");
        haset.add("Strawberry");
        haset.add("Dates");
        // Adding duplicate elements
        haset.add("Orange");
        haset.add("Mango");
        // Multiple null values
        haset.add(null);
        haset.add(null);
        // Displaying the stored HashSet elements
        System.out.println(haset);
    }
}

```

The above program produces the following output.

**[null, Strawberry, Mango, Apricot, Dates, Orange]**

The printing of the set will reveal that all values will be displayed without any order. There will only be a single iteration of both the duplicates as well as the null values that we add twice. They are treated similarly, only producing a single record in the HashSet. This is a simple test, which for some can look like the use of an array. However, HashSet implementation offers other advantages by providing methods that allow programmers to manipulate the stored object values.

It is also a good method in terms of the resources that it requires during the operations. It has good performance and works well with the handling of the load that it often possesses. The memory overhead is often not much, and the searching and rehashing operations can use more resources during its application.



Can you get a sorted data from HashSet?

#### 13.4.2.2 LinkedHashSet

Since you have understood the use of a basic HashSet, it is time to discuss the LinkedHashSet class in Java. It also contains unique elements but is different since it maintains the insertion order of the elements added in the class. The sorting of elements are important, especially when a LinkedHashSet must operate with other programming elements, where a change in order can produce a different output altogether.

The double linked list is created for all the elements. It is designed for circumstances where you need to employ an iteration order on the added object values. Using the iterator ensures that the same order is returned in which the elements are added to this hashed set, which has linked elements. Here is an example that shows the use of these practices.

```

package java11.fundamentals.chapter13;
import java.util.LinkedHashSet;
public class LinkedHashSetExample {
    public static void main(String[] args) {
        // Creating a LinkedHashSet for String
        LinkedHashSet<String> lhset1 = new LinkedHashSet<String>();
        // Adding different elements to the LinkedHashSet
        lhset1.add("Z");
        lhset1.add("R");
        lhset1.add("M");
        lhset1.add("O");
        lhset1.add("KKK");
        lhset1.add("EFG");
        System.out.println(lhset1);
        // Now creating a LinkedHashSet for Integer
        LinkedHashSet<Integer> lhset2 = new LinkedHashSet<Integer>();
        // Adding integer elements
        lhset2.add(95);
        lhset2.add(13);
        lhset2.add(0);
        lhset2.add(55);
        lhset2.add(33);
        lhset2.add(61);
        System.out.println(lhset2);
    }
}

```

The above program produces the following output.

[Z, R, M, O, KKK, EFG]
[95, 13, 0, 55, 33, 61]

Printing will start from Z and end with EFG as it was the last item added to the LinkedHashSet. The second LinkedHashSet follows the same method, where it starts from 95 and then continues printing to the last added value of 61. The double linking ensures that it is not possible to insert a duplicate element in this iteration scheme.

It is also possible to remove items. A removal simply updates the iteration, as another system print will find that the item is removed and the item next to it is updated in the order of the storage of the objects present in the set collection. Creating sets are different from the maps as there are no keys that can be used to direct towards specific values.

Linked collection classes use additional CPU resources and require more storage. The simple classes of HashSet and HashMap work well if there is no need to remember the order of the values that are inserted in the program. However, the LinkedHashSet is perfect when you want to maintain the organization of the order of the values, and then set up various objects that may allow you to store, share, and print information sets in your Java programs.

### 13.4.2.3 TreeSet

As already described with TreeMap, the TreeSet is a class that stores the order of the elements in an ascending order. It also allows you to include null elements, while it is also not synchronized. Although synchronization is possible by explicitly setting it up as a sorted set from the Java Collections Framework. Here is a simple example to show how it works before we discuss some of this class's implementation advantages in specific situations.

```

package java11.fundamentals.chapter13;
import java.util.TreeSet;
public class TreeSetExample {
    public static void main(String args[]) {
        // Creating a String type TreeSet
        TreeSet<String> tset = new TreeSet<String>();
        // Adding various string elements to the above TreeSet
        tset.add("EFG");
        tset.add("Stores");
        tset.add("Tests");
        tset.add("Pens");
        tset.add("Ink");
        tset.add("Jane");
        // Displaying the collection of TreeSet elements
        System.out.println(tset);
    }
}

```

This program will print the strings in an ascending order. The print line will display the following.

[EFG, Ink, Jane, Pens, Stores, Tests]

This shows that printing produces the display in ascending order. This function not only works well with String objects, it is also perfect for use in a variety of Integer objects environments where the order of the collection is of paramount importance. As you can see, it does not allow for repeating values as this will make it impossible to use a sorting scheme.

It is possible to specify the sorting order that you would like to use with this collection class. Also, it cannot hold objects that belong to different primitive types. This means that if a TreeSet has string values in it, it will throw a class exception when a code attempts to add an integer to the collection and vice versa.

Since this tree set can use a comparator to set up the order of the elements, it can be useful in a variety of conditions where the data is randomly collected. However, its use requires that the elements are sorted and are available for use according to a specific order, which is possible by using a comparator when defining the collections class.

### 13.4.3 List Interface

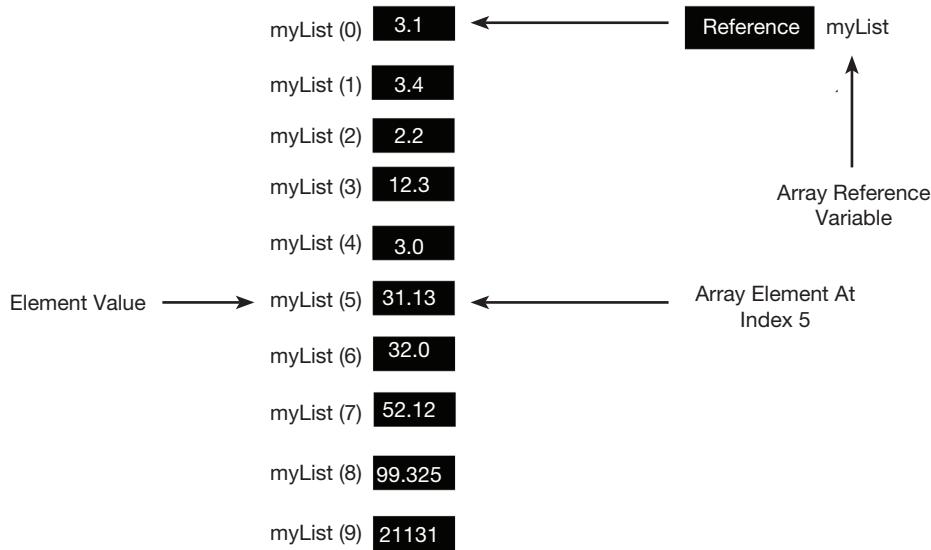
List is all about index. It offers various methods related to index, such as get(int index), indexOf(Object o), and add(int index, Object obj). List implementations are ordered by index. An object added to List will get added to the end of the list unless you specify a specific index position. Figure 13.6 shows the implementations of List.



**Figure 13.6** Implementation of List.

#### 13.4.3.1 ArrayList

The ArrayList is also a part of Java Collections. It is designed to allow programmers to use dynamic array collections. It is obviously slower than using simple arrays, but it is perfect for use when you have to perform multiple array manipulations. It belongs to the AbstractList class and therefore employs the list interface. Figure 13.7 shows the representation of ArrayList.



**Figure 13.7** Representation of ArrayList.

This class is designed to provide random access to the list of array elements. It cannot be used for primitive types. It is designed for use in the same manner as vectors are used in C++ programming language. It can contain duplicate elements as well, while it also maintains the insertion order of inputted values.

Similar to other collection classes, ArrayList is a non-synchronized class. Random access is made possible because the array operates on the basis of index, which eliminates the need to iterate to reach each item. All object manipulations are slow because there is a strong shifting which occurs when a single element is removed from the ArrayList. This list is easy to create and the available methods allow programmers to add and remove elements, while also ensuring that it is possible to use the index values of the stored array elements.

Here is a program that shows how this class can be implemented in your Java program:

```

package java11.fundamentals.chapter13;
import java.util.*;
import java.io.*;
public class ArrayListExample {
    public static void main(String[] args) {
        // First setting the size of ArrayList
        int size = 5;
        // Now declaring ArrayList with that size
        ArrayList<Integer> arrlist = new ArrayList<Integer>(size);
        // Now Appending new elements in the list
        for (int i = 1; i <= size; i++) {
            arrlist.add(i);
        }
        System.out.println(arrlist);
        arrlist.remove(2);
        // Again displaying the ArrayList after removing
        System.out.println(arrlist);
    }
}

```

This is a simple program, which will first store the first five integers in the ArrayList by the name of “arrlist”. The first print will include [1, 2, 3, 4, 5]. These values are stored sequentially just like a normal one-dimensional array indexing. This means that removing the index 2 will remove the third value from the list, which is 3 in this example. The next print will show the following output.

[1, 2, 3, 4, 5]
[1, 2, 4, 5]

This shows that it is easy to use, manipulate, and set up for different functions. Setting up array lists with fixed initial capacity is the ideal way to go about it. It ensures that you have greater control over your program and can reliably use your lists to create inputs for other program elements.



Is ArrayList safe to use in a multithreaded environment?

### 13.4.3.2 Vector

The Vector class in Java belongs to the List hierarchy, and works like a growing array of objects. All the stored objects are accessible using an integer index. It is named as a vector as the size of this class varies according to the items that are currently present in the Vector. It provides storage management by setting up a fixed capacity as well as allowing a capacity increment to ensure that it is always possible to store new objects in it.

The advantage of using this class is that it is synchronized and therefore, can be used in all types of Java programs without ever worrying about parallel processing problems. In fact, the dynamic array created in this class often contains legacy methods, which go beyond the scope of the Java Collections Framework.

Vector is great for use, if you are programming for predictable situations where you do not know the size of the required arrays. It is great for situations where you may want arrays that can smartly change their size and always use only the minimal program sources. There are various methods present in this class, which allow programmers to carry out the intended programming. Here is an example:

```
package java11.fundamentals.chapter13;
import java.util.*;
public class VectorExample {
    public static void main(String[] args) {
        // Setting up initial size and increments
        Vector vec = new Vector(3, 2);
        System.out.println("Initial size is: " + vec.size());
        System.out.println("Initial capacity is: " + vec.capacity());
        // Adding elements
        vec.addElement(new Integer(1));
        vec.addElement(new Integer(2));
        vec.addElement(new Integer(3));
        vec.addElement(new Integer(4));
        System.out.println("The capacity after four additions is: " + vec.capacity());
        vec.addElement(new Double(6.55));
        System.out.println("Now capacity is: " + vec.capacity());
        vec.addElement(new Double(5.35));
        vec.addElement(new Integer(8));
        System.out.println("Now capacity is: " + vec.capacity());
        vec.addElement(new Float(9.5));
        vec.addElement(new Integer(10));
        System.out.println("Now capacity is: " + vec.capacity());
        vec.addElement(new Integer(11));
        vec.addElement(new Integer(12));
        System.out.println("First element is: " + (Integer) vec.firstElement());
        System.out.println("Last element is: " + (Integer) vec.lastElement());
        if (vec.contains(new Integer(3))) {
            System.out.println("Vector contains 3.");
        }
        // enumerate the vector elements
        Enumeration vecEnum = vec.elements();
        System.out.println("\nElements in the vector:");
        while (vecEnum.hasMoreElements()) {
            System.out.print(vecEnum.nextElement() + " ");
        }
        System.out.println();
    }
}
```

The above program will produce the following output.

```

Initial size is: 0
Initial capacity is: 3
The capacity after four additions is: 5
Now capacity is: 5
Now capacity is: 7
Now capacity is: 9
First element is: 1
Last element is: 12
Vector contains 3.

Elements in the vector:
1 2 3 4 6.55 5.35 8 9.5 10 11 12

```

This is a detailed program that uses several ways in which the vector is created and employed. We first set up a vector with a capacity of 3 and an increment of 2. However, when we initially find the size of the vector, it is zero due to the absence of any stored value but still showing that it can accept three elements.

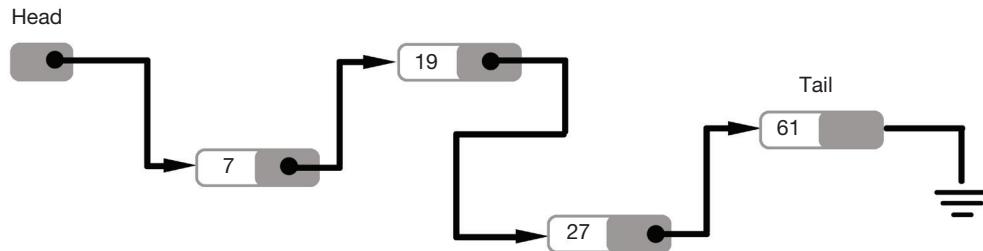
We add four elements in the Vector, which will put it above its initial capacity. This invokes an increment of 2, which is shown when the next output of the Vector capacity will show 5 as the number. The capacity is printed two more times, where it will show 7 and 9 with 2-place increments as set up.

We then show that it is possible to find the first and last element that will be stored in the same order in which we have stored them, as there is no hash function performed on them. It is also possible to check for specific elements within the vector using a test performed with the `contains()` method. An enumeration can be employed to read the values from the vector. This can then be displayed on the console.

Vector has its special application when we want to use multiple threads. However, since it is synchronized it can be slow in terms of adding and removing elements, as well performing a vector element search. It is good practice to mention the increment that you want, otherwise the default increment is to double the initial size, which can be wasteful in terms of resources.

### 13.4.3.3 LinkedList

Java provides another class from the List interface of `LinkedList`. This class provides double linking to store the elements. It can contain duplicate elements, while having the capacity to maintain the insertion order. As we have observed with similar classes, it is not designed for synchronized use. It is quick to manipulate as no object shifting is required with the double linked listing structure. Figure 13.8 shows representation of `LinkedList`.



**Figure 13.8** Representation of `LinkedList`.

`LinkedList` also implements the `Deque` interface and therefore, it can be used in a variety of ways. It is possible to use this class for stacking or creating a queue of objects. With double links, it is possible to remove the items from the start and end of the linked list. Although it is possible to add or remove items in numerous ways, access to the elements is only available in a sequential manner, where the search can occur either in a forward or reverse direction and will take time according to the position of the specific stored element.

`LinkedList` is excellent for manipulating the order of elements. This allows setting up elements for other program sections. Here is an example that shows the details of this use:

```

package java11.fundamentals.chapter13;
import java.util.LinkedList;
public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> obj = new LinkedList<String>();
        // Adding elements in various orders and positions
        obj.add("a");
        obj.add("b");
        obj.addLast("c");
        obj.addFirst("d");
        obj.add(2, "e");
        obj.add("f");
        obj.add("g");
        System.out.println("Linked list is: " + obj);
        // Now removing elements from the linked list using different options
        obj.remove("b");
        obj.remove(3);
        obj.removeFirst();
        obj.removeLast();
        System.out.println("New linked list after removing: " + obj);
        // Finding elements in the linked list
        boolean stat = obj.contains("e");
        if (stat) {
            System.out.println("List contains the element 'e' ");
        } else {
            System.out.println("List doesn't contain the element 'e'");
        }
        // Other linked list information
        int size = obj.size();
        System.out.println("Size of linked list = " + size);
    }
}

```

The above program produces the following output.

```

Linked list is: [d, a, e, b, c, f, g]
New linked list after removing: [a, e, f]
List contains the element 'e'
Size of linked list = 3

```

This is an excellent example as it shows that we can add to `LinkedList` in a variety of ways. If we add elements normally, they follow their insertion order and are indexed starting from 0. However, as already discussed, it is possible to add at the top or the bottom of the list, which will affect the entire list (e.g., we did that when we added “c” and “d”).

We also added “e” at a particular index, which will move the other items in the list by one position. Similarly, it is also possible to remove the items. We get much greater control when dealing with `LinkedList`. We can use the element value or its index position to look for the element. We can also remove the first or the last element in the `LinkedList`. The size of this list will be according to the items that we have currently stored in this collections class.

#### 13.4.4 Queue Interface

Queue, as the name suggests, holds things that need to be processed in sequence, such as FIFO or LIFO. This structure can be useful for to-do types of list. Queues offers special methods to add, remove, and review elements in addition to all the standard Collection methods. Let us see the implementation class of Queue.

### 13.4.4.1 PriorityQueue

PriorityQueue is a class that belongs to the Queue implementation, which is known for following the FIFO model. However, there are times where we need to perform processes or prioritize the elements. This is possible by using the PriorityQueue Class, which allows developers to set up priority processing operations.

This class is perfect for creating programs that may have to serve based on the priority of the available information. An example would be in a financial situation, where a company wants its premium customers to receive their processed information first. The PriorityQueue class implementation in Java will work perfectly in such collection scenarios.

This class does not allow for null values, as they are not in line with the function of this collection. It also does not allow you to create objects that cannot be compared with each other since this is essential for setting up the priority. This class uses the natural order of the elements, or employs the comparator that you have set up for the queue.

The smallest element will always be the head (the first element) of this class according to the required ordering. If there is a tie for a specific position, the class will arbitrarily place the elements, so it is best to avoid ties altogether. The queue retrieval operations also work by accessing the element at the head of the queue. It has methods from several structure trees including ones from Object, Collection, AbstractCollection, and AbstractQueue. This class is not thread-safe, but a multithreading version of the class is available for use these days. Here is an example to show how this Java Collection class is used:

```
package java11.fundamentals.chapter13;
import java.util.*;
public class PriorityQueueExample {
    public static void main(String[] args) {
        // Creating the empty priority queue
        PriorityQueue<String> prQueue = new PriorityQueue<String>();
        // Now adding the items
        prQueue.add("C");
        prQueue.add("Java");
        prQueue.add("Python");
        prQueue.add("C++");
        // Printing the most priority element
        System.out.println("The head value by using peek function is: " + prQueue.peek());
        // Now printing all elements
        System.out.println("The total queue elements are:");
        Iterator itr1 = prQueue.iterator();
        while (itr1.hasNext()){
            System.out.println(itr1.next());
        }
        // Now removing the top priority element (or head of queue)
        // And printing the modified PriorityQueue
        prQueue.poll();
        System.out.println("After removing an element with poll function: ");
        Iterator<String> itr2 = prQueue.iterator();
        while (itr2.hasNext()) {
            System.out.println(itr2.next());
        }
        // Removing one value of Java
        prQueue.remove("Java");
        System.out.println("after removing Java with remove function:");
        Iterator<String> itr3 = prQueue.iterator();
        while (itr3.hasNext()){
            System.out.println(itr3.next());
        }
        // Checking a particular element in the PriorityQueue
        boolean a = prQueue.contains("C");
        System.out.println("Does this priority queue contains C: " + a);
    }
}
```

The above program produces the following output.

```
The head value by using peek function is: C
The total queue elements are:
C
C++
Python
Java
After removing an element with poll function:
C++
Java
Python
after removing Java with remove function:
C++
Python
Does this priority queue contains C: false
```

Here, we set up a priority queue which holds the names of the programming languages. Then, we remove an element from the head of the list. This means that now the available values are rearranged. We then remove a specific value as well, and then run a search to check for an item to be present in the collection.

There are various situations in which these priorities can be really helpful during their use in the program. They are perfect for implementing logical decisions as well; for example, when requiring a reading of objects from an available collection.

### 13.4.5 Stream API

Java has an updated Stream API, which has been significantly improved over its previous version. It is designed to help developers improve their set of aggregate operations, which require the use of a sequence of objects. Stream API is a sequence of operations, where each operation can be set up in the stream. It is designed to produce a stream of objects according to the parameters that are set up in a filter element.

Object references are passed on in a stream, which result in producing a total weight of the objects. Stream operations can belong to various primitive types such as integer, long, and double. They use a pipeline that contains a source, which can be a collection of the several classes that we have discussed above, or it may constitute other elements. It is a lazy implementation where computation is only performed as and when required.

The use of stream is quite similar to a collection but is different in terms of their objectives. Collections are designed to improve the operational control required for managing the elements within them. There are terminal operations that are available for use in this API. Stream operations are great in terms of allowing the program to describe its source material and then defining the required computer operations that must be performed on them.

The iterator method is also available for use. It is possible to run queries on stream sources, while they also allow programmers to perform concurrent modifications. Java streams are designed to provide a sequence of elements on which operations can be performed on demand, differentiating them from the collections that are available in the programming language.

Stream provides both operations of pipelining and internal iterations. Here is a simple example of how different objects and collections can be passed to a stream and then work with some operations:

```
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);
```

This stream records the array as a list, and then the stream allows it to be captured in terms of the operations that are required, such as bringing out the first stored item. There are several methods that are helpful when employed with the streams in Java. Take the example of forEach, which works like a loop in the programming language.

The `forEach` loop is great at running an iteration of the items that must belong to a particular category or fulfill a specific condition from the group mentioned within the statement. This loop uses an internal iterator and can employ the interface of a `Consumer` for improved functionality. Here is an example:

```
package java11.fundamentals.chapter13;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

public class StreamExample {

    public static void main(String args[]){
        List<String> names = new ArrayList<>();
        names.add("Harry");
        names.add("Steve");
        names.add("Adams");
        names.add("Chris");
        names.add("Allen");
        names.forEach(new Consumer<String>() {
            public void accept(String name) {
                System.out.println(name);
            }
        });
    }
}
```

The above program produces the following output.

Harry
Steve
Adams
Chris
Allen

This adds various names and then carries out a loop which accepts and prints out the names from the list. This is a convenient loop which we can employ in place of a “for” loop, while it can cater to the different ways in which we can employ collections in the program loops. There are other methods that are available in collections and streams.

The list classes have various useful methods that can be combined when employed with object streams. It is possible to use the `removeIf()` method, which is designed to remove an object only if a certain condition is met. Sorting is also possible. These are all present in the `java.util` library and it is ideal to include `import java.util.*;` to ensure that complete functionality will be possible in a Java snippet or a complete program.

## 13.5 | List of Key Methods for Arrays and Collections

The methods covered in following subsections are key methods you will be using frequently in your programs. You should remember these methods so that you can find solutions to the problems you may face while using Arrays and Collections.

### 13.5.1 Arrays (`java.util.Arrays`)

Table 13.6 lists methods that are useful for Arrays. These methods can be used to work on data using Arrays.

**Table 13.6** List of Arrays Methods

Method	Description
static List<T[]> asList(T[])	This method is helpful for converting and binding Array to a List.
static int binarySearch(Object[], key)	This method is useful for searching a sorted array for a key. This returns an index of the value.
static int binarySearch(primitive[], key)	
static int binarySearch(T[], key, Comparator)	This method is useful to search a Comparator sorted array for a given value.
static boolean equals(Object[], Object[])	This method is useful to check if two arrays contain equal content or not.
static boolean equals(primitive[], primitive[])	
public static void sort(Object[])	
public static void sort(primitive[])	This method is useful to sort the array on which it is used by natural order.
public static void sort(T[], Comparator)	This method takes Comparator to sort the elements of an array.
public static String toString(Object[])	
public static String toString(primitive[])	This method prints the content of an Array in String form.

### 13.5.2 Collections (java.util.Collections)

Table 13.7 lists methods that are useful for Collections. These methods can be used to work on Collections.

**Table 13.7** List of Collections methods

Method	Description
static int binarySearch(List, key)	
static int binarySearch(List, key, Comparator)	This method is similar to Arrays, where it searches a sorted list for a given key. It then returns an Index. The overloaded version of this accepts Comparator parameter to search and returns an index of the value.
static void reverse(List)	This method is useful to reverse the order of elements in a given List.
static Comparator reverseOrder()	
static Comparator reverseOrder(Comparator)	This method is useful to accept Comparator to reverse the current sort sequence and return this Comparator back.
static void sort(List)	
static void sort(List, Comparator)	This method is useful to sort the given list by natural order. The overloaded method accepts a Comparator to sort.

### 13.5.3 Key Methods for List, Set, Map, and Queue

Table 13.8 lists key methods you will encounter periodically while working with List, Set, Map, and Queue.

**Table 13.8** List of key methods for List, Set, Map, and Queue

Method	List	Set	Map	Description
boolean add(element)	X	X		This method is useful to add an element to a Collection. In case of list, an overloaded method is available which allows to add an element at a specific index.
boolean add(index, element)	X			
boolean contains (object)	X	X		This method is useful for searching a collection for an object in order to find out if the object is present in the collection or not. This is done by returning a boolean value.
boolean containsKey(object key)			X	
boolean containsValue(object value)			X	For Maps, there are two overloaded versions of the method – one which looks for an object based on a key and the other which is based on a value.

**Table 13.8** (Continued)

Method	List	Set	Map	Description
object get(index)	X			This method is useful to get an object for a given index. In case of Map, there is an overloaded method which accepts Key to get the object.
object get(key)			X	
int indexOf(object)	X			This method returns the location of the object in a List.
Iterator iterator()	X	X		This method returns iterator for a List or a Set. Please note that Map does not have this method.
Set keyset()			X	This method is useful for Map to return Set of Map's keys.
put(key, value)			X	This method is useful to add a key/value pair to a Map.
remove(index)	X			This method is useful to remove an element for a given index. There is an overloaded method for Set which accepts object as parameter to remove. Map has an overloaded method which accepts key to remove an element from Map for the specified key.
remove(object)	X	X		
remove(key)			X	
int size()	X	X	X	This method returns the number of elements in a collection.
object[] toArray()	X	X		This method returns a collection of elements in an array form.
T[] toArray(T[])				

The following are some of the operations that are possible on Lists in Java:

1. **void add(int index, object o):** As the name of the operation suggests, this method is used to insert elements into the list. It should, however, be noted that even when elements are inserted anywhere in between the list, there is no chance of overwriting since all other elements are shifted to make room for the newly added element in the list. The index in the parameter of the method indicates the location where the new element needs to be added to the list.
2. **boolean addAll(int index, Collection c):** All of the elements present in the Collection, c, being inserted into the parameters of the method will be added to the list for which the method is being invoked. Again, the elements that were already present in the list are shifted to accommodate the elements that are to be added, ensuring that elements are not overwritten. If the list for which the operation is being invoked changes, it will return true. Otherwise, the value will be false.
3. **Object get(int index):** This operation will return the object that is stored at the location of the index that is input as a parameter.
4. **int indexOf(Object obj):** This operation will return the first instance of the instantiation of the object in the list where it is being invoked. In case the object that was input as a parameter was not a part of the list that invoked it, -1 will be returned to the user as an indication that the object does not exist from where it was invoked.
5. **int lastIndexOf(object obj):** Similar to the operation mentioned in point 4, the lastIndexOf( ) operation returns the last instance of the object of the list where it is being invoked. In case the object does not exist as an element of the list, -1 will be returned to the user as an indication that something is wrong.
6. **ListIterator listIterator( ):** Whenever this operation is called, an iterator is returned to the beginning of the list for which the operation was being invoked.

### 13.5.3.1 Comparable Interface

`Collections.sort()` uses the Comparable interface to sort Lists. Similarly, `java.util.Arrays.sort()` uses it to sort arrays of objects. This interface requires the implementing classes to implement the `compareTo()` method. The following example will help you to understand this better.

```

package java11.fundamentals.chapter13;
public class Food implements Comparable{
    private String item;

    @Override
    public int compareTo(Object o) {
        if(o instanceof Food) {
            Food f = (Food) o;
            return item.compareTo(f.getItem());
        }
        return 0;
    }
    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
}

```

The above class implements Comparable interface and implements `compareTo(Object o)` method to compare two food objects. However, with this plain implementation, we have to check if the incoming object is `instanceof Food` and then we perform `compareTo` on `String`. `String` implements `compareTo` internally so it is easier to just compare on `String` attributes of objects. However, we can improve this by using the Generics version as follows:

```

package java11.fundamentals.chapter13;
public class FoodWithGenerics implements Comparable<Food>{
    private String item;

    @Override
    public int compareTo(Food f) {
        return item.compareTo(f.getItem());
    }

    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
}

```

In the above example, using Comparable with Generics reduces the code significantly. This also guarantees that the incoming object is definitely of `Food` type.

Table 13.9 shows the return value for `compareTo()`.

**Table 13.9** Return value for `compareTo()`

Condition	Result
<code>currentObject &lt; compareToObject</code>	negative
<code>currentObject == compareToObject</code>	zero
<code>currentObject &gt; compareToObject</code>	positive

### 13.5.3.2 Comparator Interface

As we have seen earlier, there are two sort methods, one is the `Collections.sort()` that takes List, and the other one is the overloaded method that takes a List and Comparator. This interface gives the ability to sort a given collection in many different methods. Another benefit of this interface is that it allows sorting of any class even if we cannot modify it. This is contrary to Comparable, where we need to modify the class to implement the `compareTo()` method. This is a great advantage, as many times we want to sort objects based on multiple methods and we also do not have access to the source its code. Just like Comparable interface, the Comparator interface is also very easy to implement, which offers a single method `compare()`. The following code will help you to understand this better.

```

package java11.fundamentals.chapter13;
import java.util.ArrayList;
import java.util.List;
public class FoodWithComparator{
    private String item;

    public static void main(String args[]) {
        List<Food> junkFoodItems = new ArrayList<Food>();

        //Create a list of Food
        junkFoodItems.add(addItemToFoodList("Pizza"));
        junkFoodItems.add(addItemToFoodList("French Fries"));
        junkFoodItems.add(addItemToFoodList("Milk Shake"));
        junkFoodItems.add(addItemToFoodList("Burger"));
        junkFoodItems.add(addItemToFoodList("Fried Chicken"));

        System.out.println("Before Sorting : " + junkFoodItems.toString());

        //Create an object for Comparator to sort by item
        SortForComparator sfc = new SortForComparator();

        junkFoodItems.sort(sfc);

        System.out.println("After Sorting : " + junkFoodItems.toString());
    }

    public static Food addItemToFoodList(String item) {
        Food f = new Food();
        f.setItem(item);
        return f;
    }

    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
}

```

The above class uses a custom comparator that we have created as shown below. Also note that we are using the Food class which we have created in the earlier in Comparable example in Section 13.5.3.1.

```
package java11.fundamentals.chapter13;
import java.util.Comparator;
public class SortForComparator implements Comparator<Food>{
    @Override
    public int compare(Food f1, Food f2) {
        return f1.getItem().compareTo(f2.getItem());
    }
}
```

As you can see, we use food item to sort. This method is similar to the `compareTo()` method. In fact, in that method, we used the `compareTo()` method from String implementation to compare two strings. Executing the above program gives the following result.

```
Before Sorting : [java11.fundamentals.chapter13.Food@7a79be86, java11.fundamentals.chapter13.Food@34ce8af7, java11.fundamentals.chapter13.Food@b684286,
java11.fundamentals.chapter13.Food@880ec60, java11.fundamentals.chapter13.Food@3f3afe78]
After Sorting : [java11.fundamentals.chapter13.Food@880ec60, java11.fundamentals.chapter13.Food@34ce8af7, java11.fundamentals.chapter13.Food@3f3afe78,
java11.fundamentals.chapter13.Food@b684286, java11.fundamentals.chapter13.Food@7a79be86]
```

The program produces an interesting output. Although the program has sorted the objects based on the items' natural order, we are not able to see them. Can you guess why we see this instead of Food item names? This is because we have not overridden the `toString()` method and we are using it from the base class Object, which only prints Class name followed by @ and hashCode. You can easily fix this. For a reminder on how to do so, go to the Section 13.2.4.1 "Overriding `toString()` Method".

The following is the updated Food Class with overridden `toString()` method.

```
package java11.fundamentals.chapter13;
public class FoodWithToString implements Comparable{
    private String item;

    @Override
    public int compareTo(Object o) {
        if(o instanceof FoodWithToString) {
            FoodWithToString f = (FoodWithToString) o;
            return item.compareTo(f.getItem());
        }
        return 0;
    }
    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }

    public String toString() {
        return this.item;
    }
}
```

Now, we need to update the Comparator implemented class and program, which is given below.

```

package java11.fundamentals.chapter13;
import java.util.Comparator;
public class SortForComparatorWithFoodWithToString implements
Comparator<FoodWithToString>{
    @Override
    public int compare(FoodWithToString f1, FoodWithToString f2) {
        return f1.getItem().compareTo(f2.getItem());
    }
}

```

And now the program can use this newly created SortForComparatorWithFoodWithToString class for sorting.

```

package java11.fundamentals.chapter13;
import java.util.ArrayList;
import java.util.List;
public class FoodWithComparatorWithFoodToString{
    private String item;

    public static void main(String args[]) {
        List<FoodWithToString> junkFoodItems = new ArrayList<FoodWithToString>();

        //Create a list of FoodWithToString
        junkFoodItems.add(addItemToFoodList("Pizza"));
        junkFoodItems.add(addItemToFoodList("French Fries"));
        junkFoodItems.add(addItemToFoodList("Milk Shake"));
        junkFoodItems.add(addItemToFoodList("Burger"));
        junkFoodItems.add(addItemToFoodList("Fried Chicken"));

        System.out.println("Before Sorting : " + junkFoodItems.toString());

        //Create an object for Comparator to sort by item
        SortForComparatorWithFoodWithToString sfc = new
SortForComparatorWithFoodWithToString();

        junkFoodItems.sort(sfc);

        System.out.println("After Sorting : " + junkFoodItems.toString());
    }

    public static FoodWithToString addItemToFoodList(String item) {
        FoodWithToString f = new FoodWithToString();
        f.setItem(item);
        return f;
    }

    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
}

```

Let us run this program and analyze the output.

```
Before Sorting : [Pizza, French Fries, Milk Shake, Burger, Fried Chicken]
After Sorting : [Burger, French Fries, Fried Chicken, Milk Shake, Pizza]
```

From the above program, you can see how easy it is to use Comparator to sort collections.

### 13.5.3.3 Comparable versus Comparator

Table 13.10 shows the difference between these the interfaces Comparable and Comparator.

**Table 13.10** Difference between Comparable and Comparator

Comparable	Comparator
int firstObject.compareTo(secondObject) >Returns the following: firstObject < secondObject – Negative firstObject > secondObject – Positive firstObject == secondObject – Zero  The Class that needs a sorting must implement Comparable. Hence, it needs to be modified.  With this, only One sort sequence is possible.  This is implemented by Java's core classes like String, Wrapper, Date, Calendar, etc.	int compare(firstObject, secondObject) >Returns similar values as Comparable  The Class that needs a sorting need not be modified. Comparator allows building a separate class to sort desired elements. This class can later be used to sort the elements in any class.  Since we are creating a separate class to sort, we can create as many classes as we want. Hence, we can create many sort sequences.  This is intended to be used by third-party classes to sort instances.

## Summary

Java improves the facilities of Generics and collections that were present in previous versions. It significantly enhances the capabilities of the classes that are available for creating customized set of objects. Programmers need to learn the concept of generic programming.

Generics present in Java provide the facility required for efficient programming. Java offers internal Generics as well as generic methods that allow programmers to improve their coding performance.

Java also provides a detailed collections framework. This framework is designed to allow programmers to create specific collections of objects and pass them around when using different Java APIs and other functional elements. The framework provides various interfaces and describes how these interfaces are often introduced by the collection classes, which are available for implementation in the programming language. We have described these collection classes, which you can use in a variety of applications.

There are various implementations of List, Map, Set, and Queue that all have their advantages when employed for keeping a record of the objects.

We also discussed stream API, which allows for an alternate way of creating useful connections and for using them to get the job done in a variety of environments.

In this chapter, we have learned the following concepts:

1. Generic programming and how to use it in a program.
2. Generic methods and uses.
3. Collections in Java and their benefits.
4. Collection interface and implementation of collection classes.

In Chapter 14, we will explore error handling, logical errors, semantic errors, try-catch-finally block, and checked versus runtime exceptions.

## Multiple-Choice Questions

---

1. Which of the following is the most apt reason for using generics?
  - (a) Generics makes the code faster.
  - (b) Generics are useful for adding stability to the code by making bugs detectable during runtime.
  - (c) Generics are useful for making the code more readable and optimized.
  - (d) Generics are useful for adding stability to the code by making bugs detectable at compile time.
2. Which of the given parameters is utilized for a generic class to return and accept a number?
  - (a) V
  - (b) N
  - (c) T
  - (d) K
3. \_\_\_\_\_ permits us to invoke a generic method as a normal method.
  - (a) Inner Class
  - (b) Type Interface
  - (c) Interface
  - (d) All of the above
4. \_\_\_\_\_ consists of all the collection classes.
  - (a) Java.awt
  - (b) Java.util
  - (c) Java.net
  - (d) Java.lang
5. What do you understand by a Collection in Java?
  - (a) A group of classes
  - (b) A group of interfaces
  - (c) A group of objects
  - (d) None of the above

## Review Questions

---

1. How are Generics useful?
2. What are the different collection interfaces?
3. Which collection class is used to store key-value pair data?
4. What is the difference between ArrayList and Vector?
5. Explain stream API.
6. What is LinkedList? How it is useful?
7. What are implementing classes of Map interface?
8. When is Set useful? How do we use it?

## Exercises

---

1. Create a chart to show all the collection interfaces and implementation classes.
2. Write a program that demonstrates the use of all the collection classes.
3. Use a real life example to write a program to implement PriorityQueue.

## Project Idea

---

Create a program to add and display non-persistent data of all the vehicles entered into a parking lot. Non-persistent data means that we cannot use the database to store this data and hence we have to hold this data in memory. The data should be captured in a way that we can segregate vehicle-specific and driver-specific information separately but linked to each

other. Finally, display this information on a web page and the admin should be able to search the data based on various features such as color of the car and registration number.

**Hint:** Use various collection classes to hold this data in memory.

## Recommended Readings

---

1. Philip Wadler, Maurice Naftalin. 2010. *Java Generics and Collections: Speed up the Java Development Process*. O'Reilly Media: Massachusetts
2. The Java™ Tutorials, Oracle: <https://docs.oracle.com/javase/tutorial/java/generics/index.html>
3. The Java™ Tutorials, Oracle: <https://docs.oracle.com/javase/tutorial/collections/intro/index.html>



# Error Handling

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Error handling.
- Logical errors.
- Syntactical errors such as capitalization, string split, missing or improper imports of classes, and missing curly braces.
- Semantic errors such as improper use of operators, incompatible types, precision, and scoping.
- Importance of error handling with try, catch, and finally blocks.
- Checked versus runtime exceptions.

### 14.1 | Introduction

Error handling is an important concept in all programming languages. It must be learned and implemented with accuracy to ensure that problems do not ensue in the long run. Even though most people are intimidated by error handling and believe it to be daunting and complicated process, the fact remains that this process is imperative for efficient and effective programming.

Tossed around often, the term “error handling” essentially refers to an entire world of anticipating the possibility of code errors, creating mechanisms to detect them and ultimately, resolving these errors and anomalies in code using various programming and communicating processes. To help put things into perspective, we will first describe error handling from a programmer’s point of view to understand how significant error handling is actually for creating and implementing a well-structured program or piece of code.

Next, we will shed some light on the different terms that are involved in error handling and exception handling situations including try, catch, and finally. Then, we will differentiate between runtime and compilation errors. We will also explain how and why the exceptions and errors in both cases vary so greatly, especially in Java programs.

Towards the end of our discussion, we will also shed some light on the different techniques that can be used for error and exception handling. We will also explain how each type of error or exception should be handled based on the situation. This will help programmers get a better understanding of how problems, inaccuracies, and anomalies in the code or program should be dealt with in a manner that solves the problem effectively and efficiently as possible.

### 14.2 | Understanding Error Handling

Error handling involves a lot more than just the removal of errors. In fact, error handling refers to the entire system of procedures, techniques, and processes that are required to anticipate and identify where problems lie before we get to deal with an error or an exception in the code itself.

Anticipating and knowing that there is a perpetual possibility of errors or exceptions manifesting themselves at any place in your code is the first step involved in effective error and exception handling. This is because once you begin to write code keeping this fact in mind, you will naturally begin to incorporate best practices in your code, minimizing the possibility of errors and exceptions altogether. When errors are anticipated and the possibility of their manifestation is kept in mind, chances are that you will also double check your code more often than you otherwise would in order to minimize problems in the future.



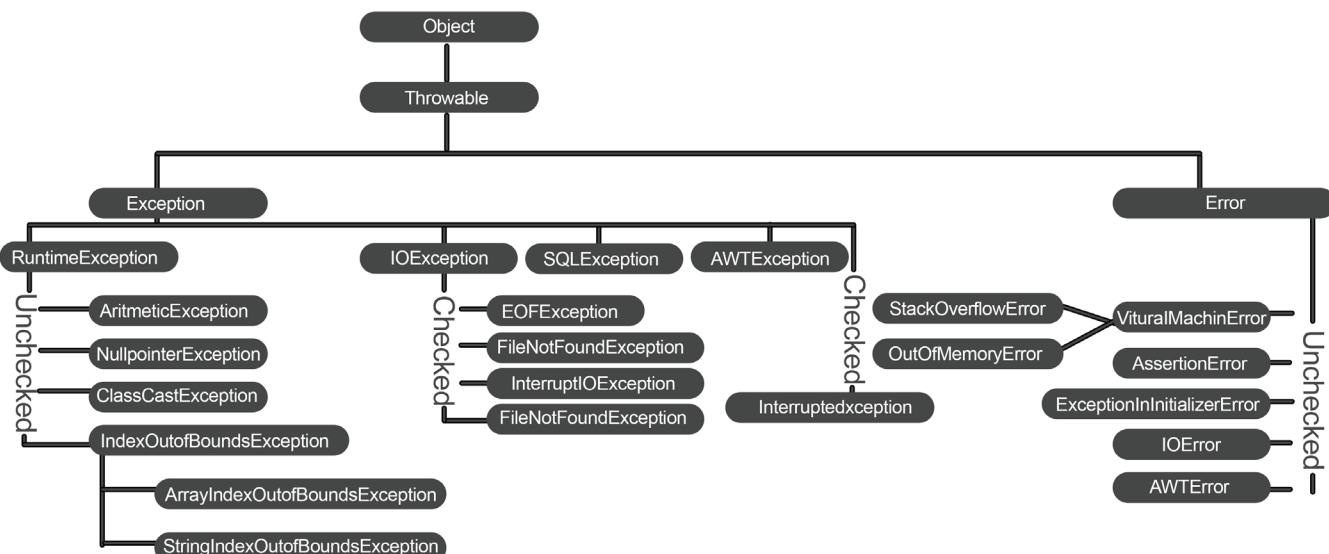
Since errors and exceptions can even occur despite checking the code multiple times, it is imperative for programming languages such as Java to have an error handling system in place, allowing developers to implement a systematic approach to resolve them.

Unlike most concepts that deal with either the software or hardware side exclusively, error handling is one of the few concepts that erases the distinguishing line between the two. Error and exception handling does not only help identify both software and hardware problems, but also allows programmers and developers to deal with the problem at hand in an effective manner, ensuring that the functionality of the rest of the program is not compromised. Needless to say, without the right error and exception handling procedures and techniques in check, it would be impossible for programmers and developers to change one block of code without affecting the rest of it in the process.

Fortunately, in today's day and age, programmers have two major options when it comes to error and exception handling. Programmers can either develop codes and programs that allow room for dealing with errors, or they can make use of software and tools available online and elsewhere to handle problems that are caused by errors and exceptions. While the distinction of the type of error is often clear, there are certain cases in which an error may seem to fall either in a number of different categories, or it may be difficult to identify a category at all due to the ambiguous nature of the error or exception. In cases like these, it is suggested that you make use of customized software for the identification of the errors and exceptions.

There are four major categories of all errors in Java and a majority of other programming languages. These categories are logical errors, generated errors, compile-time errors, and run-time errors. Needless to say, there are different techniques and processes that are involved in dealing with errors of each of these different categories. While the errors in certain categories may be solved and avoided altogether with the help of some basic proofreading of the code, other types of errors may require the programmer or developer to identify the problem with the help of test data and deal with it using resolution programs.

Figure 14.1 illustrates the exception hierarchy in Java.



**Figure 14.1** Exception hierarchy in Java.

#### QUICK CHALLENGE

Create a chart which shows the difference between Exception and Error. Also provide one example of each.

### 14.3 | Logical Errors

Logical errors are often considered to be the most difficult types of errors to spot. This is primarily because instead of causing a program to terminate or stop working altogether, logical errors and exceptions produce incorrect results. What this means is that a program with logical errors will run perfectly fine, but you will not be able to see the results which you anticipated, be it due to a typo or incorrect usage in terms of logical operator precedence.

To help you understand better the importance of keeping logical operator precedence in mind, we will illustrate the damage that may arise with the help of the example below. You can see how something as simple as the different usage of parentheses can produce different outputs of 11, 13, 9, and 8.



```

package java11.fundamentals.chapter14;
public class LogicalErrors {
    public static void main(String[] args)
    {
        // Create variables Var1 thru Var4
        int Var1 = 5 + 4 * 3 / 2;
        int Var2 = (5 + 4) * 3 / 2;
        int Var3 = (5 + 4) * (3 / 2);
        int Var4 = (5 + (4 * 3)) / 2;
        // Print the results.
        System.out.println(
            "Var1: " + Var1 +
            "\nVar2: " + Var2 +
            "\nVar3: " + Var3 +
            "\nVar4: " + Var4);
    }
}

```

The above program produces the following result.

Var1: 11
nVar2: 13
nVar3: 9
nVar4: 8

Since logical errors have more to do with the logic of a program than the actual structure, simple proofreading often suffices for identification and resolution of such errors. While proofreading your code is a practice that should always be implemented, it is essential to double check your code several times especially if it involves a lot of logic building. This is also because logical errors can cause situations and problems that are a lot more severe. In some cases, a condition that is false may even be assumed as true due to incorrect operator usage – something that has the potential to cause errors and problems that will carry on in the rest of the program.

In certain cases, logical errors that have not been spotted at an early stage in the code or program may even have the potential to affect data and values that appear thousands of lines of code after the error was initially made.

One of the most common logical errors that bother programmers is misuse or misplacing of a semicolon. While misplaced semicolons may not give you the results that you had intended, the fact remains that it is completely possible to create a fully functional Java code with misplaced semicolons.

Here is an example of how misplaced semicolons work:

```

package java11.fundamentals.chapter14;
public class ErrorForLoop {
    public static void main(String[] args)
    {
        // Variable Declaration.
        int Counter;
        // Create For Loop.
        for (Counter = 1; Counter <= 10; Counter++)
        ;
        {
            // Print the result.
            System.out.println("Counter is " + Counter);
        }
    }
}

```

The above program produces the following result.

Counter is 11

In the above example, the only output that will be displayed to the user will be “Counter is 11”. This is because the semicolon was placed right after the creation of the *for* loop, instead of placing it at the end of the code block. Had the semicolon been placed at the end of the entire *for* loop block, the output would be a list of individual values for Counter starting from 1 and finishing at 11.

As already mentioned, most logical errors can be removed by proofreading, where you go through each line of code and make sure that all the coding elements are properly set up to avoid such errors. In fact, it is also possible to avoid them in most integrated development environments (IDEs) that provide color recognition of different program elements. Programmers can quickly find out where they may have left a code situation that will result in a logical error.

Techniques that are employed for error handling are often termed as *debugging* or *troubleshooting*. Runtime errors are also quite significant in regard to *debugging* or *troubleshooting*, and they are often handled by setting up countermeasures in the programming environment to avoid such situations altogether. In fact, applications that run on hardware are often designed to have an error handling method, which allows the application to recover from any error and once again restart to provide functionality to the program.

## 14.4 | Syntactical Errors



Syntactical errors are errors in which the wrong language, or syntax, is used to write a code or program. Not writing the condition in parentheses for an *if* loop, for instance, would classify as a syntax error since that is a requirement for the code to run properly. This even holds true if the condition were to be present on the same line as the *if* statement, but just not in parentheses.

Syntactical errors may not be as difficult to spot as their logical counterparts because the compiler will most likely catch the majority of these errors for you. However, as always, proofreading and knowing the correct syntax and conditions for all loops and codes is essential to ensure that you do not have to face any problems in the long run.

The reason why it is so important to spot and resolve syntactical errors as early as possible is because if a code or program has syntactical errors, it will not be possible for the Java Runtime Environment (JRE) to use the byte code that needs to be created by the compiler. Since syntactical errors have the potential to cause a lot of problems, we are sharing some of the most common syntactical errors and why they are so important to solve.

### QUICK CHALLENGE

Write a program that can demonstrate syntactical errors.

### 14.4.1 Capitalization

Java is a case-sensitive language but not a lot of new programmers realize that. This is the major reason why so many new Java programmers start capitalizing keywords instead of writing them in lowercase. Capitalization may or may not make a difference in certain languages; however, in languages such as Java, making a change as apparently insignificant as writing myCount instead of MyCount has the potential to ruin the entire code and fill it with more errors than you ever thought would be possible.

Using the right capitalization is essential for all class names, variable names, and any other piece of code that you will be writing in Java.

### 14.4.2 Splitting Strings

Dividing your code into a number of lines often does not matter when you are coding in Java. However, there is an exception that applies when you are adding strings in your program or code. Splitting a string so that it comes on more than one line or contains a new line character in it will cause the compiler to throw an exception or object to the code that you have written.

Fortunately, there is a way to create strings that absolutely have to be written in multiple lines. The approach that is required to do this without receiving any errors messages, exceptions, or objections is to add a double quote to the string that appears on

the first line, and then add a plus sign right after this first half of the string ends to show the compiler that whatever follows in the next one or more lines needs to be added or concatenated to the same string. An example of how this can be done is as follows:

```
System.out.print ("This is the first half of the string " +
"this is the second half of the string that needs to be concatenated. " );
```

### 14.4.3 Not Importing Classes

One of the most common semantic errors that programmers – both new and seasoned – make when coding in Java is forgetting to include an associated class when they wish to make use of a particular API feature. For instance, if you wish to incorporate the String data type in your code, it is essential to add the class to your application using `Import java.lang.String;` for the String class to be imported in your application.

### 14.4.4 Different Methods

When coding in Java, you must remember that static methods and instance methods work differently in this language. Static methods are those that are associated with a specific class, whereas instance methods are associated with the object that is created from a certain class. In case you treat a static method as an instance method in Java, your compiler will present you with a syntactical error since the way in which both of these are dealt with differ greatly.

### 14.4.5 Curly Braces

When programming in Java, you will often find yourself in situations where you want the same feature to apply to more than one-line code. In cases like these, it is imperative for you to create a block of code enclosed in curly braces to ensure that the compiler understands where the code that the feature needs to be applied to starts and finishes. While the compiler may catch this error for you in most cases, it is still important for you to keep an eye out for lines and blocks of code that need to be treated as a single entity and make sure that they are distinctly identifiable by the compiler.

For instance, if you forget to finish the contents of a class with a curly braces, this will be treated as a syntactical error by the compiler and you will be notified of a missing curly braces. In the example below, the class Cat will not be recognized because the compiler will not know where it ends.

```
package java11.fundamentals.chapter14;
public class Cat {
    int age;
    String breed;
    String color;
    void meowing() { }
    void sleeping() { }
    void hungry() { }
}
```

As seen in the above example, each of the methods – namely meowing, sleeping, and hungry – do not have any contents in them but they still have curly braces to show where they start and end. The class Cat, on the other hand, does not end with a curly braces which is why an error or exception will be generated.

Moreover, in the scenario mentioned above, while the compiler knows that a curly braces is missing, it may not be able to pinpoint the exact location where the curly braces should appear. This is because each of the methods may or may not have been a part of the Cat class. This is why you will only be notified that a curly braces is missing but not where it should appear.

Since using a curly braces in Java is the syntactical rule when a feature or action needs to be applied to multiple lines of code, runtime errors can also occur in case you forget to add a curly braces or you add one in the wrong spot. In case you forget to add a curly braces at the end of an `if` statement, the condition will only apply to the line of code that comes immediately after the `if` statement and can potentially cause problems in the way the application or program was intended to run.



## 14.5 | Semantic Errors

Figuring out the difference between semantic and syntactical errors is one of the biggest problems for both beginners and seasoned programmers of the Java language. While the majority of people tend to classify both semantic errors and syntactical errors in the same category due to ease and convenience, there are certain significant differences between these types of errors. While syntactical errors have to do with the syntax of the code, semantic errors are related to the usage of the code. This means that it is possible for you to have semantic errors in your code even if the syntax is correct.

While you might have probably already guessed this, the most common type of semantic errors are those in which variables are used without proper initialization. You already know that a variable cannot be used or be expected to add value to your code in case there are problems with its declaration or its initialization. Fortunately, these errors will be caught by the compiler in most cases and you will receive a notification about the variable in question.

While the problem with variables is the most common semantic error, there are plenty of others that you should be aware of as a programmer of the Java language. Subsequent subsections discuss some of the most common semantic errors and explain why each of them occurs. In some cases, we may also tell you how they can be solved or avoided altogether.



What is the difference between syntactical errors and semantic errors?

### 14.5.1 Improper Use of Operators

Sometimes, in case of operators are used improperly on variables, it may give an impression of syntactical error. However, in reality it is more of a semantic error. For example, the increment operator (++)<sup>4</sup>, for instance, cannot use Boolean variables and attempting to do so will be a semantic error.

While new versions of Java are able to detect these problems far more easily, finding out exactly why an error message is generated may be a bit of a challenge especially if you are not sure what operators are allowed to be used with what variables.

Another common operator error mistake the use of comparator (==) operator with objects. Since this is not allowed and comparator operators can only be used with primitive types, an error message will be generated showing that the operation intended is not permissible.

### 14.5.2 Incompatible Types

Whether it is done by accident or simply due to the lack of knowledge, programmers of the Java language often try to use incompatible types together. Needless to say, doing so is a semantic error that may or may not be caught by the compiler.

For instance, if you mistakenly try to assign a float value to an int variable, the compiler will present an error message. However, if you try to assign an int to a float, the compiler will automatically convert the integer value into a float value. This could potentially cause several problems, especially if that conversion was never intended. Additionally, since the programmer or user will not be notified of this conversion as the compiler will practically expect that this was the desired output, it will become almost impossible for the anyone working on the code to find out that the code contains an error.

### 14.5.3 Precision

While a float variable can be converted to an int variable by applying casting, doing so incorrectly can result in a loss of precision. Additionally, since everything after the decimal will automatically be lost, the precision of the value that you use will be affected along with the results in all of the other lines or blocks of code where the value in question will be used. It is, therefore, recommended that you only use casting when you absolutely must and know that your output can potentially be affected.

### 14.5.4 Scoping

Scoping is an issue that tends to bother even the most experienced programmers. This is because there are quite a number of rules that define what is and is not allowed within a certain scope. For instance, if you try to declare a private static int variable inside a method, you will receive an error. Instead, you should declare the variable globally so that it can be used properly. Following programs are examples of incorrect and correct ways to declare a private static int variable.

If you have a class called *VariablePrivate* and you declare the private static int variable globally in the class itself, it will work as follows:

```
package java11.fundamentals.chapter14;
public class VariablePrivate {
    // This is the correct way to declare the private static int variable called
    intPrivate.
    private static int intPrivate = 5;
    public static void main(String[] args)
    {
        // The contents of the method will go here
        System.out.println("intPrivate value : " + intPrivate);
    }
}
```

The above program produces the following result.

intPrivate value : 5

On the other hand, if you try to declare the private static int variable called *intPrivate* within the method itself, you will get an error message. Following is the incorrect way of declaring the private static int variable:

```
package java11.fundamentals.chapter14;
public class VariablePrivateIncorrect {
    public static void main(String[] args)
    {
        private static int intPrivate = 5;
    }
}
```

The above program produces the following result.

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
  Illegal modifier for parameter intPrivate; only final is permitted
  at java9.fundamentals.chapter6.VariablePrivateIncorrect.main(VariablePrivateIncorrect.java:10)
```

## 14.6 | Importance of Error Handling

The importance of error handling can never be emphasized enough. Error handling does not only ensure smooth operation but also guarantees that the code will not malfunction and will provide the desired results.

Since even the simplest of applications are easily a few thousand lines of code long and comprise code written by a number of programmers, it is extremely important to take care of all errors and exceptions as you move forward. This will ensure that errors, exceptions, and other problems of the sort do not carry forward until the end of the program.

### 14.6.1 Try, Catch, and Finally

Earlier, we talked all about errors and their different types along with ways in which they can be avoided or resolved. By now, you probably already understand the importance of error handling and realize that it is imperative to resolve errors as soon as they are spotted to ensure that they do not continue to cause problems in the rest of the code. While error handling is something that you are probably already aware of, there is another thing that you should probably worry about – exceptions.

So, what exactly are exceptions and how do they work? As the name suggests, exceptions derange the regular flow of the program or code and make it act in a way that it should not. While there is nothing wrong with the syntax of the code in which an exception is being caused, exceptions still cause problems and make your code or application act in a way that is different than what was expected.

In Java, the concept of exceptions is a lot more profound. Error events in Java are wrapped by exceptions that occur within a method. Exceptions in Java do not only contain information about the error that occurred and the type of the error that has manifested itself in the code, but also details the state of the program when the error occurred. Additionally, some other custom details that can help you assess the nature of the error and the impact that it caused on the code may also be included in the exception.

Fortunately for Java programmers, exceptions can point out a multitude of different error conditions that may occur within the code. When talking about Java virtual machine (JVM) errors, exceptions cannot only help indicate OutOfMemory errors and StackOverflow errors, but they can also help the programmer understand Linkage errors and why they occurred within the code along with details about the error itself. Exceptions can also help programmers understand System errors, including FileNotFoundException exceptions, IOExceptions, and SocketTimeOutExceptions.

The reason why many programmers of Java language are interested in using exceptions is primarily because these allow them to treat the regular flow of a code as a separate entity, unlike error handling. As a result, you do not only get clearer algorithms that are far easier to handle and understand than regular code, but the clutter within the code also decreases significantly, allowing you to be more creative and innovative with your programs and applications.



Can you delay the exception handling further down the method calls?

In programs or pieces of code where an exception is possible, it is recommended that you use a statement that will be able to catch the exception. By doing so, you can prevent the entire program from crashing should the exception occur. One of the most common statements used for this purpose is the “try” statement. By incorporating a try statement in your code, you can be sure that the potential exception will be caught and treated as a block of code that is separate from the remaining program, code, or application that it is a part of. The general form of the try statement is as follows:

```
try
{
// statements that can potentially cause an exception will go here
}
catch (identifier of type of exception)
{
//statements that should be executed if the exception is thrown will go here
}
```

As seen above, the try statement helps you treat a block of code with the potential of an exception as a separate entity to ensure that the rest of the program is not affected. Additionally, it is seen that the try statement also accommodates an alternative statement or block of code that should be executed should the exception be thrown, as seen in the second half of the example above.



Can you capture errors using try-catch?

In try-catch statements, it is possible for you to code multiple try blocks. This particularly comes in handy where the statements in the try block throw exceptions of different types. Additionally, from Java 7 onwards, it is even possible to catch multiple types of exceptions within the same catch block by separating the type using vertical bars. An example of how this is done is as follows:

```
try
{
    // statements that have the potential to throw
    // ClassNotFoundException
    // ArrayIndexOutOfBoundsException
}
catch (ArrayIndexOutOfBoundsException | ClassNotFoundException e)
{
    System.out.println(e.getMessage());
}
```

It is also important to note that the contents of the try block are not visible to the catch block, which is why it is impossible for any variables declared in the try block to be used in the catch block or blocks. In case there is a variable that you need to use in both blocks of code, it should be declared before the try block.

The following is another example of how the try-catch statement can be used in action to prevent the user from trying to divide the value of a variable by 0.

```
package java11.fundamentals.chapter14;
public class DivideByZero {
    public static void main(String[] args)
    {
        int var1 = 5;
        int var2 = 0; // 0 is assigned to var2 to cause an exception by dividing var1 by 0
        try
        {
            int var3 = var1 / var2; // This is the statement that will cause the exception
        to be thrown
        }
        catch (ArithmaticException e)
        {
            System.out.println("It is not possible to divide by zero");
        }
    }
}
```

The above program produces the following result.

<b>It is not possible to divide by zero</b>
---

Another block of code that is used for exceptions in Java is the finally block. One unique and interesting feature about the finally block is that it is always executed regardless of whether or not any exceptions have been thrown in the code. With that said, using the finally statement is among best practices and is expected to be used particularly in scenarios when you are closing a connection or file. Here is an example of a program where the finally block will be executed:

```

package java11.fundamentals.chapter14;
public class FinallyBlockExample {
    public static void main(String args[])
    {
        try
        {
            int var = 30 / 6;
            System.out.println(var);
        }
        catch (NullPointerException e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("These are the contents of the finally block");
        }
        System.out.println("The finally block has been executed");
    }
}

```

The above program produces the following result.

5  
These are the contents of the finally block  
The finally block has been executed

Since no exception will be thrown, the value of the variable “var” will be displayed to the user, after which the finally block will be executed and the words “These are the contents of the finally block” will be displayed. Next, the words “The finally block has been executed” will be displayed to the user.

## 14.7 | Checked verses Runtime Exceptions

As the name suggests, checked exceptions are those that are identified at the time when the code is being compiled. On the other hand, runtime exceptions are identified when the code is being run.



Can program execution continue even after an exception is thrown?

### 14.7.1 Checked Exceptions

In case a checked exception is being thrown by a method, it is necessary that this exception should either be handled by the method itself, or the *throws* keyword should be used to specify it. The following example shows how a checked exception is thrown in the *main()* function.

```

package java11.fundamentals.chapter14;
import java.io.*;
public class CheckedExceptionsExample {
    public static void main(String[] args)
    {
        FileReader file = new FileReader("D:\\\\newfolder\\\\example.txt");
        BufferedReader fileInput = new BufferedReader(file);
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());
        // This block of code will output the first 3 lines of the file
        // "D:\\newfolder\\example.txt"
        fileInput.close();
    }
}

```

The above program produces the following result.

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
  Unhandled exception type FileNotFoundException
  Unhandled exception type IOException
  Unhandled exception type IOException

  at java9.fundamentals.chapter6.CheckedExceptionsExample.main(CheckedExceptionsExample.java:9)
```

In the example above, the `main()` function uses `FileReader` to read the file located at `D:\\newfolder\\example.txt`. (If you are executing this code on your end, make sure you change the file location as per the location of file on your computer). However, a checked `FileNotFoundException` is thrown by `FileReader()`, whereas checked `IOException` is thrown by the `close()` and `readLine()` methods. As a result, a message will be displayed showing where the exception was thrown, explicitly stating that the source code cannot be compiled.

### 14.7.2 Runtime Exceptions

Runtime exceptions in Java are a subcategory of unchecked exceptions or exceptions that are not checked during the compilation of the code. Since unchecked exceptions and runtime exceptions are not detected at the time of compilation, the code compiles perfectly before the runtime exception is detected when the program is run. The following is an example of a runtime exception:

```
package java11.fundamentals.chapter14;
public class RuntimeExceptionExample {
    public static void main(String args[])
    {
        int var1 = 0;
        int var2 = 10;
        int var3 = var2 / var1;
    }
}
```

The above program produces the following result.

```
Exception in thread "main" java.lang.ArithmaticException: / by zero
  at java9.fundamentals.chapter6.RuntimeExceptionExample.main(RuntimeExceptionExample.java:9)
```

In the example above, the `main()` function will throw an `ArithmaticException` which is a type of runtime exception, as it is not possible to divide by 0. However, since there is nothing wrong with the syntax of the code, it will compile perfectly before the exception is detected when the program is run.

#### QUICK CHALLENGE

Write a program which can demonstrate all types of Runtime Exceptions.

## Summary

In this chapter, we have discussed error handling and how to use it. We also discussed the concept of error handling and elucidated various types of errors such as logical, syntactical, and semantic. Then we discussed the importance of error handling in which we studied the use of try, catch, and finally blocks. At the end of the chapter, we learnt the difference between checked and runtime exception with examples.

In this chapter, we have learned the following concepts:

1. What is error handling? What is the importance of using it?
2. What are logical errors, syntactical errors such as capitalization, string split, missing or improper imports of classes, and missing curly braces?
3. What are semantic errors such as improper use of operators, incompatible types, precision, and scoping?
4. How do we handle errors using try, catch, and finally blocks?
5. What is the difference between checked and runtime exceptions?

In Chapter 15, we will learn about garbage collection, using it in a program, and how to implement the garbage collector.

## Multiple-Choice Questions

---

1. Exceptions arises during \_\_\_\_\_ in the code sequence.
  - (a) Compilation time
  - (b) Run time
  - (c) Can occur anytime
  - (d) None of the above
2. \_\_\_\_\_ is not an exception handling keyword.
  - (a) finally
  - (b) thrown
  - (c) catch
  - (d) try
3. Exception can be thrown manually by using \_\_\_\_\_ keyword.
  - (a) finally
4. Which of the following is the parent of Error?
  - (a) Object
  - (b) Collections
  - (c) Throwable
  - (d) Exception
5. What do you understand by unchecked exceptions?
  - (a) Checked by Java virtual machine
  - (b) Checked by Java compiler
  - (c) (a) and (b)
  - (d) None of the above

## Review Questions

---

1. What is error handling?
2. How is error handling useful?
3. What is the difference between error and exception?
4. Which exception is thrown when no class is found?
5. How do try, catch, finally blocks work?
6. What are semantics errors?
7. What are logical errors?

## Exercises

---

1. Write a program that can produce exceptions and catch them using try, catch, finally blocks.
2. Create a comparison chart to distinguish between error and exception.
3. Write a program that can produce errors. Observe and document the outcome.

## Project Idea

---

Create a calculator program that performs various types of arithmetic operations. Also create a program that can divide any number by any number. Make sure you add exception

handling to capture and notify users in case they use unpermitted operations, such as dividing a number by 0. Make sure your program has all the features of a normal calculator.

## Recommended Readings

---

1. Oracle Tutorials – [https://www.w3schools.com/java/java\\_tryCatch.asp](https://www.w3schools.com/java/java_tryCatch.asp)
2. W3Schools – <https://docs.oracle.com/javase/tutorial/essential/exceptions/>
3. Oracle Technetwork – <https://www.oracle.com/technetwork/java/effective-exceptions-092345.html>

# Garbage Collection

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Java memory management.
- Garbage collection.
- How to code according to the garbage collector.
- How to make objects eligible for collection.
- The latest updates of garbage collection.

### 15.1 | Introduction

Memory management is always an extremely important part of any program or application being developed in the Java programming language. Whenever objects are created for classes in any programs or applications developed using Java language, they are allocated a certain amount of memory, allowing the program or application to function the way that it should.

Just like objects in Java, variables are also allocated memory that allows them to be stored and used throughout the program. All variables and objects need to be assigned or allocated certain areas of memory to ensure that they can be used without errors or exceptions manifesting themselves. However, the memory that is allocated to them differs according to the scope that the said variable is located in.

Once the memory allocated to a certain variable or object is deemed useless and the purpose of the variable or object has been completed, it is important for the allocated memory to be recycled and be usable for other purposes. And that is where the garbage collector in Java comes into play.

Fortunately for developers of the Java programming language, memory reclamation is automatic in the Java virtual machine (JVM), which means that Java developers do not necessarily have to go out of their way in order to free memory objects that are no longer being used. Garbage collection in Java also works based on the assumption that objects are short-lived and can be easily reclaimed once they have been created.

Another plus in Java is that if there are ever any objects that are not referenced, they are automatically removed from heap memory to free up space for other objects and variables, making this an extremely memory-efficient language.



What will happen to your program at runtime if there is no Garbage Collection taking place?

### 15.2 | Garbage Collection in Java

Since garbage collection has to do with memory management more than anything else, it is imperative for us to talk about how memory works in Java programming. It is clear that garbage collection in Java is an automatic process, and that the programmer or developer creating a piece of code or developing an application does not need to go out of their way in order to tell the machine which objects in the code need to be deleted. However, there are certain specifications that need to be met to ensure that the object will be deleted. And this is where memory management is important.

For the sake of simplification, let us say that the memory heap for Java is divided into three major sections – Young Generation, Old (or Tenured) Generation, and Permanent Generation.



Whenever a new object is created, it is located in the Young Generation. This Young Generation is further divided into two subcategories – Eden Space and Survivor Space. Upon creation, new objects are present in the Eden Space and are moved to Survivor Space 1 after the first garbage collection cycle.

A minor garbage collection event then follows in order to move objects from the Young Generation to the Old Generation. The Old Generation contains all objects that have matured enough to be moved from here but cannot fall in the category of Permanent Generation. When the garbage collector needs to remove objects from the Old Generation, this is known as major garbage collection event.

All data that is required for proper functioning of the application or code is stored in the Permanent Generation. This generation, therefore, stores meta data and information regarding classes. In case there is a class that does not need to be used, it can be removed from the Permanent Generation with the help of a garbage collector in order to free memory for other classes or data that is imperative for the code to run properly.

Most developers regard the Permanent Generation as a block that is contained in the native memory instead of the heap memory. Since the Permanent Generation contains class definitions by class loaders, this block has inherently been designed to expand and grow to ensure that there are no out of memory errors or exceptions that are thrown in the code. However, in case the block needs more memory than available in the physical memory, the operating system ensures that virtual memory is made available for the code to run like it should. This virtual memory will certainly allow the code to run; however, in order to make use of this virtual memory, the constant back and forth will be required between the virtual memory and physical memory. This affects the performance and smoothness of the code.

Now that you have a basic understanding of memory heap and how it works, we can start talking about the process that is involved in garbage collection. A daemon thread is created and used by the JVM for garbage collection. Whenever a new object is created, the JVM attempts to get the space that is required for the object from the Eden Space. As is the rule, the Survivor Spaces and Tenured Space are empty at the beginning of the code.

In case the JVM is unable to find the required memory from Eden Space, minor garbage collection is initiated to free up the required space. For this process, one of the two Survivor Spaces, S0 or S1, are regarded as the To Space. Next, all objects that are not reachable are copied by the JVM to the To Space, and 1 is added to their age. On the other hand, all objects that are not fit for the Survivor Space are moved to Tenure Space.

Since not every object is meant to move from the Young Generation Space to the Tenured Space, JVM comes with a Max Tenuring Threshold. This is basically an option that can be modified according to the preferences of the programmer or the requirements of the application to ensure that there is always enough memory for the creation and initiation of new objects. While the default value of the Max Tenuring Threshold is set as 15, it can be changed.

As mentioned earlier, a minor garbage collection process occurs in order to reclaim memory that can be freed from the Young Generation (when objects become mature and move on to Tenured Space). It is important to note that garbage collection is a Stop The World process in Java, which means that the garbage collector ensures that all the threads that are being used to run the application or program are stopped and only the threads that are being used for garbage collection are still running until the process is complete. It is also important to keep in mind that Stop The World will occur regardless of the algorithm that is being used for garbage collection.

The number of threads being used for garbage collection will depend on the algorithm that is being used for the process. Based on the algorithm, garbage collection could either be done successfully using a single thread or multiple different threads working together to clean out memory. Additionally, while the delay caused by the STOP-THE-WORLD application is often negligible, in cases where there is a lot of memory to be cleaned, garbage collector tuning can also be applied to reduce the STOP-THE-WORLD time.



Can we guarantee garbage collection?

## 15.3 | Major Garbage Collection

If minor garbage collection occurs very frequently, the objects from the Young Generation will naturally move into Tenured Space and occupy all of the available memory very quickly. Since that will prove to be detrimental to the program or application, JVM will trigger a major garbage collection event. While major garbage collection is also referred to as full garbage collection at times, it should be noted that full garbage collection entails reclaiming memory from the Meta Space as well.

And while this is one way in which major garbage collection can be triggered, there are a number of other possibilities as a result of which JVM can call major garbage collection. Even though it is generally advised against, if a programmer decides to

call `Runtime.getRuntime().gc()` or `System.gc()`, the JVM will trigger a major garbage collection. It is also possible for major garbage collection to be triggered if there is not enough memory remaining in the Tenured Space, if the JVM is unable to reclaim the required amount of memory from the Eden Spaces or Survivor Spaces, or if enough space is not available for the JVM to load new classes or objects as they are created in the program or application.



Is there any situation where the garbage collector stops working?



## 15.4 | G1 and CMS Garbage Collectors

Java offers various different types of garbage collectors, which have their own advantages and disadvantages. It is important to learn about garbage collectors to understand which one to use for your specific needs. The following are the two most significant garbage collector options offered by Java:

1. **Garbage First (G1) garbage collector:** Introduced in Java 7, G1 is capable to handle very large heaps efficiently and concurrently. For Java 9, this is the default garbage collector. If you are using a version prior to Java 9, you may enable G1 with `-XX:+UseG1GC` parameter for JVM. G1 offers various advantages:
  - (a) Uncommitting unused heap.
  - (b) Free up memory space without using a long pause time.
  - (c) Work concurrently such as without interrupting or stopping application threads.
  - (d) Deal with very large heap by using non-continuous spaces.
  - (e) It can collect both young and old generation spaces at once. This can be achieved by G1 by splitting the heap into hundreds of small regions instead of only three (i.e., Eden, Survivor, and Old) like most other garbage collectors do.

Mainly, GC1 outshines other garbage collectors on large amount of data as it does not have to work on the entire heap or entire generation. It can simply work on the selected small regions and finish quickly. On the disadvantage side, GC1 struggles to work with small heaps.

2. **Concurrent Mark Sweep (CMS) garbage collector:** As the name suggests, Concurrent Mark Sweep uses multiple threads (“Concurrent”), which are used to scan through the heap and mark the unused objects (“Mark”) that can be collected and recycled (“Sweep”). Many applications strive for shorter garbage collection pauses and do not get affected by sharing their processor resources with the garbage collector while the application is running. These are the perfect candidates to use CMS. Also, the benefits of this garbage collector are for the applications which got a large set of long-lived data (such as a large tenured generation) and execute on multiprocessors. You can enable the CMS collector with the following command-line option:

`-XX:+UseConcMarkSweepGC`

There are a few challenges in using CMS collector, such as finding the right time to initiate the concurrent work, as this work can get completed before the application is out of available heap space. CMS requires higher percentage of heap space than Parallel garbage collector in a scale of 10% to 20%. Hence, it is a costlier proposition for using shorter garbage collector pause times. Another challenge is related to handling the fragmentation in the old generation. When old generation goes through the garbage collector process, it may occur that the free space between objects get smaller or nearly non-existent. Hence, the objects which are getting promoted from the young generation do not find sufficient place to fit. Since CMS concurrent collection does not do any type of compaction, whether incremental or partial. This unavailability of space for promoted objects forces CMS to a full collection using Serial garbage collector. This results in a lengthy pause.

### QUICK CHALLENGE

Write a memory-intensive program which creates a lot of objects. Try G1 and CMS collector on this program. Print timestamp and heap size. Use the following commands to print the heap size and free space.

Command to print total memory of heap:

```
Runtime.getRuntime().totalMemory()
```

Command to print free memory of heap:

```
Runtime.getRuntime().freeMemory();
```

## 15.5 | Advantages of Garbage Collection in Java



Garbage collection in Java is essential for its benefits pertaining to freeing up memory to ensure that it can be used for other purposes. However, limiting the advantages of garbage collection to just that is an injustice of the highest degree. Since you are not responsible for keeping tabs on the data and figuring out when it is no longer necessary for a certain object or needs to be cleaned after being left behind by an object that is not referenced anymore, you do not only have the time to deal with everything else that is on your plate, but automation of the garbage collection process also makes you more productive.

And that is not all. Since manual garbage collection in Java has been made extremely difficult, it is possible for programmers or developers to accidentally cause the program or application that they are working on to crash due to incorrect removal of certain objects from memory. Additionally, since removal and updating of memory works automatically in the programming language, integrity of the program is maintained at all times.

## 15.6 | Making Objects Eligible for Garbage Collection



The only reason why an object will be suitable for garbage collection in Java is if the reference variable of the object is no longer available. Objects that fall under this category are also termed as *unreachable objects*.

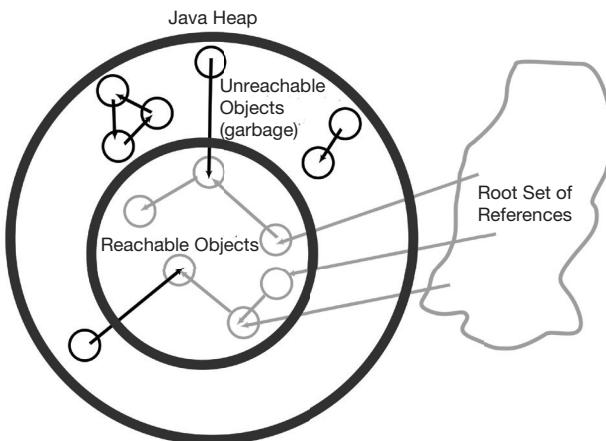
While there are a number of different ways in which an object can be made eligible for garbage collection, it is important to first talk about the reference of an object and how referencing works. By now, you are probably aware that every object is assigned a certain amount of memory from the available heap, but the operation behind the process is slightly complicated. Every time an object is automatically assigned memory through the operator, a reference to that memory is returned back. For ease of understanding, this reference is basically the address of the newly created object in the memory that is given to it by the “new” operator.

At this point, it is important to note that if an object is being referenced, it is not necessary that it is being used at that particular moment in the program or application. What this means is that if an object is passed as an argument or assigned to a variable, this action essentially only takes the reference of the object into account and does nothing more. An example of this type of referencing is as follows:

```
Book myJavaBook = new Book();
```

### 15.6.1 Unreachable Objects

As mentioned earlier, only objects that no longer have a reference associated with them can be deemed or made eligible for garbage collection. Why is that the case? This is because an object that does not have a reference is essentially not present on the memory heap, which means that it is not available for use and cannot add any value. Figure 15.1 shows the unreachable and reachable objects.



**Figure 15.1** Reachable and unreachable objects in Java heap.

Since we are talking about how objects can be made eligible for garbage collection, the first scenario where this process can be applied is when objects are created within the scope of a method. Whenever a method is called, it is pushed on or moves directly to the stack that contains all methods that are important for the successful execution of the program or application. Now, when this method is popped or removed from the stack, all of the members that were associated with this method die with it.

In case there were any objects that were created in this method, they will also die off, leaving unreferenced objects on the heap that can no longer be used for any sort of value addition. Based on the premise that they do not have any reference, these anonymous objects automatically become eligible for garbage collection.

Here is a program to demonstrate that objects that are created within the scope of a method will be deemed useless after execution of that method is complete.

```
package java11.fundamentals.chapter15;
public class UnreachableObjectsExample {
    private String myObject;
    public static void main(String args[])
    {
        // Executing testMethod1 method
        testMethod1();
        // Requesting garbage collection
        System.gc();
    }
    public UnreachableObjectsExample(String myObject)
    {
        this.myObject = myObject;
    }
    private static void testMethod1()
    {
        // After existing testMethod1(), the object myObjectTest1 becomes unreachable
        UnreachableObjectsExample myObjectTest1 = new
        UnreachableObjectsExample("myObjectTest1");
        testMethod2();
    }
    private static void testMethod2()
    {
        // After existing testMethod2(), the object myObjectTest2 becomes unreachable
        UnreachableObjectsExample myObjectTest2 = new
        UnreachableObjectsExample("myObjectTest2");
    }
    @Override
    protected void finalize() throws Throwable
    {
        // following line will confirm the garbage collected method name
        System.out.println("Garbage collection is successful for " + this.myObject);
    }
}
```

Since both the objects within the method had become unreachable, the output will be as follows.

<b>Garbage collection is successful for myObjectTest2</b>
<b>Garbage collection is successful for myObjectTest1</b>

The output shows that any object within a method becomes useless after execution of the method; the object automatically becomes eligible for garbage collection.

## 15.6.2 Reassigning Reference Variables

Reference IDs are extremely important in Java and help in addressing each of the objects and variables that are being used in any code. In case, one object's reference ID is used to refer to other object's reference ID, then the first object that was initially being referenced becomes unreachable and cannot be used in the program or code in any way. Once it becomes unreachable due to the reference ID being used for multiple objects, the first object is deemed eligible for garbage collection.

The following program is an example of a situation where a reference ID is used to reference multiple objects.

```
package java11.fundamentals.chapter15;
public class ReassigningReferenceExample {
    private String myObject;
    public ReassigningReferenceExample(String myObject)
    {
        this.myObject = myObject;
    }
    public static void main(String args[])
    {
        ReassigningReferenceExample testObject1 = new
        ReassigningReferenceExample("testObject1");
        ReassigningReferenceExample testObject2 = new
        ReassigningReferenceExample("testObject2");
        // testObject1 now refers to testObject2
        testObject1 = testObject2;
        // Requesting garbage collection
        System.gc();
    }
    @Override
    protected void finalize() throws Throwable
    {
        // following line will confirm the garbage collected method name
        System.out.println("Garbage collection is successful for " + this.myObject);
    }
}
```

Since the reference ID of the first object, `testObject1`, is eventually being used to reference the second object, `testObject2`, the first object becomes unreachable and is suitable for garbage collection, as shown in the code above. The output of the code will be as follows.

**Garbage collection is successful for testObject1**

The example above shows the importance of using the right reference ID at the right time to ensure that objects or variables that are crucial for the successful execution of your code, application, or program are not uselessly lost.

## 15.6.3 Nullified Reference Variables

Another extremely effective method to make an object suitable for garbage collection is making all of the variables that reference to it NULL. When this is done, you will have a scenario similar to the one mentioned above, and the object will have no references to it, essentially making it useless or unreachable. As soon as the object becomes unreachable, it is suitable for garbage collection, and the garbage collector can be called to remove it from the heap.

The following is an example code that shows how nullifying the reference variables of an object can make it unreachable and eligible for garbage collection:

```

package java11.fundamentals.chapter15;
public class NullifiedReferenceVariablesExample {
    private String myObject;
    public NullifiedReferenceVariablesExample(String myObject)
    {
        this.myObject = myObject;
    }
    public static void main(String args[])
    {
        NullifiedReferenceVariablesExample testObject1 = new
        NullifiedReferenceVariablesExample("testObject1");
        // Setting testObject1 to Null will qualify it for the garbage collection
        testObject1 = null;
        // Requesting garbage collection
        System.gc();
    }
    @Override
    protected void finalize() throws Throwable
    {
        // following line will confirm the garbage collected method name
        System.out.println("Garbage collection is successful for " + this.myObject);
    }
}

```

Since there is no longer any reference to `testObject1` and its reference variable was made `NULL`, `testObject1` is no longer reachable in the code and becomes suitable for garbage collection. When the garbage collector is called, it finds `testObject1` without any reference and removes it from the heap.

The output of the code above will be as follows.

**Garbage collection is successful for testObject1**

#### 15.6.4 Anonymous Objects

Anonymous objects can be used in Java to call methods. However, what distinguishes anonymous objects from regular objects in Java is that anonymous objects do not have any reference IDs. As per the criteria, this makes anonymous objects the perfect candidates for garbage collection.

The following code is an example of a method being used on an anonymous object:

```

package java11.fundamentals.chapter15;
public class AnonymousObjectsExample {
    public static void main(String[] args) {

        System.out.println(new AnonymousObjectsExample().myMethod());

    }
    public String myMethod() {
        return "I love this book";
    }
}

```

The output of the code above will be as follows.

I love this book

Now that you understand how anonymous objects can be used to successfully call and run methods, here is an example of how garbage collectors can be used to remove anonymous objects from the heap.

```
package java11.fundamentals.chapter15;
public class AnonymousObjectsGarbageCollectionExample {
    String myObject;
    public AnonymousObjectsGarbageCollectionExample(String myObject)
    {
        this.myObject = myObject;
    }
    public static void main(String args[])
    {
        // Anonymous Object is being initialized without a reference id
        new AnonymousObjectsGarbageCollectionExample("testObject1");
        // Requesting garbage collector to remove the anonymous object
        System.gc();
    }
    @Override
    protected void finalize() throws Throwable
    {
        // following line will confirm the garbage collected method name
        System.out.println("Garbage collection is successful for " + this.myObject);
    }
}
```

Since there is no any reference to the anonymous object, the garbage collector will successfully remove it from the heap. The output of the code above will be as follows.

Garbage collection is successful for testObject1

## 15.7 | JEP 318 – Epsilon: A No-Op Garbage Collector



Interesting changes were proposed to the way garbage collection was approached in Java in a March 2018 update. For Java 11 version, the goal was to develop a garbage collection mechanism that took care of memory allocation but did not quite use a significant methodology for the reclamation of reusable memory from the heap. In this update, once the heap for Java was exhausted, the JVM would shut down. This garbage collector is called no-op garbage collector, which is also known as Epsilon.

Since the proposition for the update was based on the premise that the garbage collector would not reclaim memory from the heap, the no-op garbage collector would encourage the creation of ultra-performing applications that do not have any garbage or can do without a garbage collector for memory reclamation. The no-op garbage collector is intended to be simultaneously available with other garbage collectors and will not come into effect unless it is activated explicitly.

While this comes as an interesting update for the public, developers and researchers are particularly interested in the viability and practicality of the no-op garbage collector, considering how memory allocation and garbage collection work in Java. Moreover, since the no-op garbage collector will be simultaneously available with the other garbage collectors, users will be able to benefit from the no-op garbage collector or Epsilon collector as a control variable to gauge the performance of the remaining available garbage collectors.

To test the efficiency and effect of garbage collectors on the performance and speed of an application, variable configurations of garbage collectors can also be used simultaneously with the no-op garbage collector with the same workload. By doing so, garbage collector developers will not only be able to see how the performance of applications differs based on the configuration of the garbage collector in a controlled environment, but they will also be able to understand how garbage collectors work in a more isolated manner.

The Epsilon garbage collector or no-op garbage collector can prove to be highly beneficial for a limited number of applications and libraries that do not produce any garbage. Since the presence of a garbage collector is essentially not necessary, removing the overhead of the garbage collector can improve the efficiency of the application or library that is being used. However, to create a library that supports the Epsilon garbage collector, a number of factors including the library's memory management aspect without the use of a garbage collector must be taken into consideration. Since implementation of Epsilon garbage collector will essentially leave the application with no mechanism for reclamation of memory, it will have to be ensured that garbage is either non-existent, or minimal to the extent that the memory of the application does not run out.

When talking about the Epsilon garbage collector, it is important to consider the risks and benefits of implementing a no-op garbage collector and weighing them against the problems that will manifest in order to achieve a state of no or minimal garbage in any application or program. While there are a considerable number of difficulties in reaching a no garbage state considering how memory management works in Java, a few aspects of garbage collection need to be discussed to get an idea of how achieving such a state may be possible.

There are two major mechanisms used by JVM for memory management in Java. While most memory management operations are done through the heap, the stack is equally important in order to create an application that is equal parts memory-efficient and functional. The presence and use of both the heap and stack is the primary reason why there are two different types of errors when it comes to memory management – OutOfMemoryError and StackOverflowError.

The stack is only visible and used by threads that are running at a certain point in time, that too during the execution of its particular method. When the execution of the thread that is using the stack is complete, it leaves the stack and memory is automatically freed without the use of a garbage collector. It is the memory on the heap that needs to be checked by a garbage collector to see whether or not it can be cleaned to add value to the application or program that is being used. However, it is important to note that all 8 primitive data types go directly on the stack, which makes it possible for the application or program to run efficiently without the use of a garbage collector. If the Epsilon garbage collector is to be implemented, it is suggested that primitive data types be used for the majority of purposes to ensure that there is no any additional strain or reason for the presence of a conventional garbage collector.

Contrary to popular belief, objects can also be created without the use of a garbage collector. This means that fully-functional applications and programs can still be created with the Epsilon update with just a little additional effort.



Can we avoid OutOfMemoryError?

## Summary

With this discussion about the Epsilon update available, we close the topic of garbage collection and how it works in the Java language. It is assumed that the codes, examples, and scenarios provided have helped you understand the ins and outs of garbage collection in the Java language and will give you an idea of the different ways in which garbage collection can be approached. In this chapter, we have learned the following concepts:

1. How garbage collector works.
2. What is the use of garbage collection?
3. Importance of memory management.
4. How to make objects eligible for garbage collection.
5. What are the latest updates in garbage collection?

In the Chapter 16, we will learn about String and I/O operations. We will also explore Java's file management capabilities and tools available to read, write, and manipulate file content.

## Multiple-Choice Questions

1. At the time of object destruction, \_\_\_\_\_ method is utilized to execute some action.
 

<p>(a) delete()</p>	<p>(b) finalize() (c) main() (d) None of the above</p>
---------------------	--

2. \_\_\_\_\_ requires the highest memory.
- Class
  - JVM
  - Stack
  - Heap
3. Where does the new object memory get allotted?
- JVM
  - Young Space
  - Old Space
  - Young or Old Space, depending on space availability
4. \_\_\_\_\_ is a garbage collection algorithm which has two phases operation.
- Space management tool
  - Sweep model
  - Cleanup model
  - Mark and sweep model
5. Which of the following is not a Java Profiler?
- Jconsole
  - JVM
  - Jprofiler
  - Eclipse Profiler

## Review Questions

---

- How does garbage collection work in Java?
- How do we make objects eligible for garbage collection?
- How can you instruct the garbage collector to initiate the garbage collection process?
- What is the inner working of the garbage collector?
- Why is memory management important?

## Exercises

---

- Create a diagram that shows how objects become eligible for garbage collection.
- Write a program that initializes a lot of objects in a loop and observe how much time it takes to crash the program.
- Write a program that can generate stack overflow error. Document the findings.

## Project Idea

---

Create a voting system program that can collect the entire list of the voters from all around the country and allow them to vote. This program must validate the identity of the users.

Calculate the number of votes. Make sure you use good garbage collection practices so the program will not crash due to memory management issue.

## Recommended Readings

---

- Benjamin J. Evans, James Gough, and Chris Newland .2018. *Optimizing Java: Practical Techniques for Improving JVM Application Performance*. O'Reilly Media: Massachusetts
- Erik Ostermueller. 2017. *Troubleshooting Java Performance: Detecting Anti-Patterns with Open Source Tools*. Apress Media: New York
- Oracle Technetwork: Java Garbage Collection Basics – <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

# Strings, I/O Operations, and File Management

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- All about strings.
- Various string operations such as concatenation and split.
- StringBuffer and StringBuilder.
- I/O operations.
- InputStream and OutputStream.
- File management.
- Access files and manipulation of files.

### 16.1 | Introduction

The String class of Java programming language is one that interests many due to its unique characteristics. Unlike the strings of other programming languages such as C++ or C, which are essentially only arrays of chars, strings in Java comprise immutable sequences of Unicode characters that make them unique in many ways.

To make the creation and manipulation of strings easier than it is in most programming languages, the String class in Java offers several different features. Owing to these features, there are a few different techniques that can be used to create and modify strings according to one's requirements or preferences and to ensure that they add value to the application, program, or block of code where they are being used. The creation of strings, for example, can be done using either a string literal, or by using a constructor and calling it for the creation of a String instance. Both techniques will be explained in detail in subsequent sections in this chapter.

### 16.2 | Role of Strings in Java

Before we discuss the different reasons and ways in which Strings can be used in Java, it is important examine a few facts that every programmer of the Java language should know about strings. We already know that Strings in Java are unlike strings in other languages such as C and C++. They are not simply arrays of chars, and there is a lot that makes the Strings in Java unique.

One main feature that makes Strings in Java important is concatenation. With just a simple “+” sign, you can easily concatenate the contents of two or more strings – something which is not possible with other objects in the Java language, such as Circle or Point. Additionally, since the contents of Strings in Java is as immutable as mentioned above, modifications cannot be made to the contents of any string that is being used in any application, program, or block of code. This means that functions like `toLowerCase()` or `toUpperCase()` will create an entirely new string instead of modifying or changing the contents of the current string, and this new string will then be returned.

The use of null characters for the termination of strings in other languages such as C or C++ is another way in which Strings of Java are different. Strings in Java are objects that are backed by character arrays. If you wish to view the contents of the String in Java in the form of the character array that represents it, you can use the `toCharArray()` method of the String class.

Comparison of Strings in Java is also far easier than it is in other programming languages. Instead of following a lengthy process for the comparison of strings, all you need to do is use the `equals()` method for the comparison of two strings that are being used in any program, application, or block of code. This is possible because the String class of Java overrides the `equals()` method, making comparison of strings extremely easy, convenient, and hassle-free.

Similarly, searching for substrings within a string of Java is also far easier than in other languages. With the help of regular expressions and simple methods such as `indexOf()` and `lastIndexOf()`, parts of strings can be searched for and values

returned if a match is found. Strings can also be trimmed and split into multiple other strings using regular expressions and used separately for a variety of purposes in any application, program, or block of code in Java.

Now that you are aware of the variety of benefits and features that the String class and strings in Java language offer, we will discuss the String class and how strings work in Java programming.

You have learned about how memory is managed by garbage collection in Java language in Chapter 15. Here is a quick recap. There are two major entities that are used for memory management in Java – the heap and the stack. The stack is used for the execution of operations and processes as they are called in the block of code, program, or application. On the other hand, the heap has more to do with storage of contents that are required for the effective running and execution of the code, program, or application.

But what does that have to do with strings? The answer is not exactly simple or straightforward by any means. Strings and the String class of Java programming language are given special treatment, and any string literals that are used in the programming language are assigned a special storage space in the heap memory known as *string constant pool*. Whenever string objects are created using string literals in Java, these objects are stored in the string constant pool. On the other hand, when string objects are created using the new keyword, they are treated just like other objects and are sent to the heap for storage purposes. The following is an example of how string objects are created using string literals:

```
package java11.fundamentals.chapter16;
public class LiteralExample {
    public static void main(String args[]) {
        String message = "Hello World! I love Java";
        System.out.println(message);
    }
}
```

The String Object of the string message shown below with contents “Hello World! I love Java” has been created with the help of a string literal. It will go in the string constant pool instead of being sent to the heap memory like other string objects.

**Hello World! I love Java**

Similarly, String Objects can also be created using the new keyword as follows:

```
package java11.fundamentals.chapter16;
public class KeywordExample {

    public static void main(String args[]) {
        char[] javaArray = { 'I', ' ', 'L', 'O', 'V', 'E', ' ', 'J', 'A', 'V', 'A' };
        String javaString = new String(javaArray);
        System.out.println(javaString);
    }
}
```

The example above shows how a string object can be created using the new keyword. See the output below.

**I LOVE JAVA**

As mentioned earlier, string objects that are created using this method are treated like normal objects and are sent to the heap, where they are stored along with other objects and variables that are important for the execution of codes or programs.

What most people do not know about the string constant pool is that pool space is allocated to objects depending on the content of the string object in question. This means that when objects are sent to the string constant pool, they are checked to ensure that there are not any two objects that have the same content.

Whenever a new object needs to be created using string literal, the Java virtual machine (JVM) goes through the content of the object that the user wants to create, and then double-checks the content of the objects that are already available in the pool. If an object with content that is the same as the one that needs to be created already exists in the pool, the reference of this object is returned and the new object is not created. The new object will only be created if the content in it is unique and distinct.

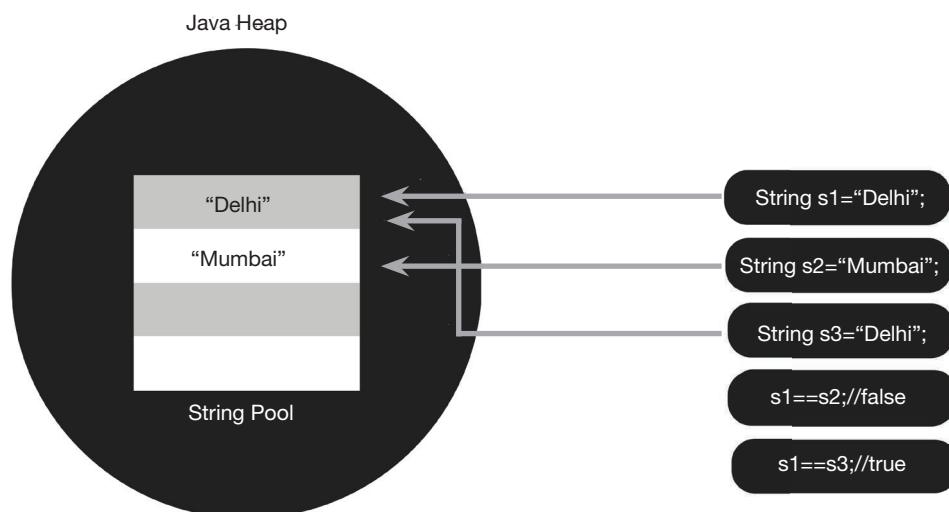
However, this is not the case when the new keyword is used for the creation of a new string. If you attempt to create a new string using the new keyword, it will be created whether or not it contains the same contents as an existing string. This shows that two string objects present in the heap memory can have the same contents, but that is not the case with string objects present inside the string constant pool. This can also be proven using the `==` operator, which only returns as true if the physical address of both objects being compared is the same.

```
package java11.fundamentals.chapter16;
public class StringObjectComparison {
    public static void main(String[] args) {
        // The following strings are being created using literals
        String literal1 = "xyz";
        String literal2 = "xyz";
        System.out.print("Comparison using == operator for literals : ");
        System.out.println(literal1 == literal2); // The output of this line of code will
be true
        // The following two strings are being created using the new operator
        String keyword1 = new String("abc");
        String keyword2 = new String("abc");
        System.out.print("Comparison using == operator for objects : ");
        System.out.println(keyword1 == keyword2); // The output of this line of code will
be false
    }
}
```

The above program creates the following output.

```
Comparison using == operator for literals : true
Comparison using == operator for objects : false
```

Since the two string objects created using string literals having the same contents will also have the same physical address, the output of the first comparison returns as “true”. On the other hand, even though the contents of the string objects created using the new keyword are also the same, this second comparison will return as “false” because each of these string objects will have different physical addresses in the heap. This further proves that two objects cannot have the same contents if they need to be stored in the string constant pool, whereas it is completely normal for them to coexist in the heap memory. See Figure 16.1 to understand how String objects are located on Java heap and how they are compared to each other.



**Figure 16.1** String Pool on Java heap and `==` Comparison.

Understanding the difference between the strings that go on the heap and those that go on the stack is imperative in order to understand which objects can potentially be garbage collected more easily, once they are no longer in use. There are essentially four major types of references that exist in Java:

1. Strong references.
2. Weak references.
3. Soft references
4. Phantom references.

For the sake of explanation, consider the following example:

```
Greeting hello = new Greeting();
```

In the line of code above, “hello” acts as a strong reference to the `Greeting()` object.



Can the garbage collector collect a String object?

In case an object does not have a strong reference (and only has a soft reference), there is a high possibility of the memory of said object being recollected in case the garbage collector needs additional memory for operations. On the other hand, if an object only has a weak reference assigned to it, the garbage collector will reclaim the memory of this object in the next cleaning phase, regardless of whether or not enough memory already exists.

If an object does not have a strong, weak, or soft reference, then the reference that it has is often called *phantom reference*. Unable to be accessed directly, these references are not known to many programmers or developers of Java, which makes them particularly interesting. Another important fact is that whenever the `get()` method is used on phantom references, they always return as null.

The most popular and powerful types of references – strong references – are used extremely common for programs, blocks of code, and applications that are developed using Java. Objects can be easily created in Java language and assigned references. It is important to note that whenever an object has a strong reference, it can never be garbage collected.

Since strings are given special treatment in Java, the same holds true when it comes to garbage collection of string objects that are used in blocks of code, programs, or applications created using the language. As you already know, every time a thread is created and started, it has its own stack memory. Here, it is important to note that even if an object is present in the heap, but is no longer being referenced by the stack, it becomes eligible for garbage collection. Even if an object in the heap has strong references to other objects present within the heap, they become eligible for garbage collection and will eventually be removed or deleted if they do not have a reference from the stack.

Here are a few facts regarding the garbage collection process and how it really works:

1. Garbage collection is an automatic process in Java. What this means is that starting the process is at the discretion of the JVM.
2. Garbage collection is actually an expensive process. This is because the running of the garbage collector essentially puts all the other threads of the application, program, or block of code on hold, until the garbage collection process is completed.

More complex than simply calling a method and freeing up memory, the garbage collection process essentially entails the use of the mark-and-sweep mechanism that helps JVM decide which objects need to be kept alive for the program, block of code, or application to run and be executed effectively. This helps us understand that even though garbage collection essentially works automatically in Java, certain objects and processes can still be left alive to ensure that the quality, efficiency, and/or performance of the application, program, or block of code in use is not being compromised in any way.



What are some of the most common string operations in the Java programming language?

## 16.3 | Types of String Operations

There is a wide variety of operations that can be applied to strings when used in Java language. As mentioned earlier, strings in Java can be concatenated and split, and they can also be formatted. Since these operations are extremely important and commonly used for the creation of codes, we will explain them in detail along with examples of codes to help you understand how exactly each operation can be used in your own applications or programs.



### 16.3.1 Concatenation

Concatenation is one of the most commonly used operations when it comes to strings in Java language. Concatenation in Java is the name of the process that is used to combine the contents of two or more strings to create a new string. There are two major methods that can be used to concatenate two strings in Java:

1. The first method involves the use of the “+” operator.
2. The second method involves the use of the concat() method of the String class.

#### 16.3.1.1 Concatenation Using the Addition “+” Operator

Concatenation using the addition “+” operator is the most commonly used technique to add the contents of two strings in Java. It is important to note that whenever you want to add two or more string literals together, they should be within double quotes.

For instance, if you want to combine the strings “Hello” and “people of the world”, you should write the line of code as follows:

```
"Hello" + " people of the world!" //result: Hello people of the world!
```

Here, it is also important to note that for your sentence to read properly, you should ensure that you properly add spaces in the double quotes.

You can also use the concatenation option for printing outputs in Java. The code to add the contents of the same two strings as mentioned above and print the output is as follows:

```
System.out.println("Hello" + " people of the world!"); //the output Hello people of the world! will be printed on the screen.
```

1. **Combination of strings on more than one line:** When it comes to string literals, Java does not accommodate the contents of a string to span multiple lines. This is another area where the concatenation option can come in handy.

With the help of the concatenation operation using the “+” operator, you can create a string literal that spans multiple lines, shown as follows:

```
String PopularQuote = "An eye for an eye" +
" will make the whole world blind."
```

2. **Concatenating variable objects:** While the “+” sign is often used as an arithmetic operator, the rules change considerably in case one of the operands with the “+” sign is a string. In such a case, the other operand is also converted into string form to ensure that it makes sense when concatenated with the operand of String type. Let us take a look at an example:

```
float weight = 50.0;
System.out.println("My weight is " + weight);
```

The output for the line of code written above would be:

```
My weight is 50.0
```

In the example above, weight is a float variable, so the “+” operator will first convert the operand to String type and then concatenate the two strings, as it normally would. Even though it is not visible to the end user or programmer who wrote the block of code, the conversion from float type to String type is done by calling the `toString()` method. This shows that concatenation operation does a lot more than just combine the contents of two strings, and that background operations will also be done if needed.

#### QUICK CHALLENGE

Consider two strings s1 and s2. Assign values to each like s1=“ABC” and s2=“XYZ”. Now try using “\*” operator on those two and note down the result.

### 16.3.1.2 Concatenation Using the concat() Method

The second technique that can be used for concatenation of two strings in Java is by using the `concat()` method. Whenever the `concat()` method of the `String` class is used, the method is applied to the first string that needs to be added to form the result, and the second string that needs to be concatenated is taken as a parameter. Let us see the following example:

```
public String concat (String myStr)
```

The line of code above shows how `concat()` method takes the second string as a parameter to add, or concatenate, it with the first string.

The following is another example of how an entire block of code is written to concatenate two strings using the `concat()` method of the `String` class:

```
package java11.fundamentals.chapter16;
public class StringConcatExample {
    public static void main(String args[]) {
        String myStr = "My Favourite Programming Language";
        myStr = myStr.concat(" is Java");
        System.out.println(myStr);
    }
}
```

The output of the block of code above is shown below.

**My Favourite Programming Language is Java**

This shows that the `concat()` method works in a way that is different from the “+” operator. There are quite a number of other differences between these two techniques.

1. While the “+” operator can be used to concatenate objects of variable types, the `concat()` method can only be used to combine objects of the `String` type. What this means is that the `concat()` method of the `String` class will only work effectively in case it is called on a variable of `String` type and has a parameter of the `String` type that needs to be concatenated.

This makes the `concat()` method a lot more limited than the “+” operator. The latter is a lot more convenient and hassle-free since it can convert variables of a number of data types into `String` type efficiently and effectively, allowing this operator to offer a wider range of benefits and usages than the `concat()` method.

2. The second major difference between these two methods is that an exception is thrown by the `concat()` method if the object that is entered as a parameter has a null reference. This means that the `concat()` method of the `String` class throws a `NullPointerException` whenever a parameter has a null reference. On the other hand, the “+” operator treats the second operand as a null string and still concatenates it with the first string or operand.
3. Unlike the “+” operator that can be used to concatenate multiple strings, the `concat()` method of the `String` class can be used for the concatenation of only two strings at a time.

Due to the reasons mentioned above, it goes without saying that the “+” operator is used more commonly for concatenation of strings in Java than the `concat()` method. However, since there are significant differences in the working of both of these techniques, the performance and efficiency of applications will also differ depending on the technique that is being used for concatenation.

### 16.3.2 Splitting Strings

Another extremely common operation that is performed on strings in Java is splitting. There are several different reasons why you might need to split a string into two or more parts, which is why it is important for you to understand the working behind the `split()` method of the `String` class.

The `split()` method of the `String` class in Java splits or divides the input string into multiple parts based on the regular expression that is entered as a parameter. The result of this operation is an array of strings that are divided according to the regular expression that was input as the parameter. In one variant of the `split()` method, you even have the option of entering your desired limit for threshold of the result that will be the output.

For the first variant of the `split()` method, the only thing that is required is the string that needs to be split. All other operations and workings will be done by the Java programming language itself.

The following is an example of how the `split()` method works without a limit:

```
package java11.fundamentals.chapter16;
public class StringSplitExample {
    public static void main(String args[]) {
        String myStr = "My Favourite Programming Language : Java";
        String[] arrOfStr = myStr.split(":");
        for (String piece : arrOfStr) {
            System.out.println(piece);
        }
    }
}
```

The result of the block of code given above as an example is shown below.

This shows that it is not necessary for one to enter any limits when using the `split()` method in the Java. In the block of code mentioned above, the String `myStr` was used as an example, and the string was split by the ":" as mentioned in the following line:

```
String[] arrOfStr = myStr.split(":");
```

When printed, the block of code gave us the desired result.

As mentioned earlier, the other variant of the `split()` method of the `String` class in Java requires the user to add a limit to the result. This limit is entered as an integer parameter. It is important to note that there are three types of values that the limit integer can take:

- 1.** Positive.
- 2.** Negative.
- 3.** Zero.

Since the value that you select for the limit has an effect on the result that you will get after the string is split, it is important to understand how the value of the result can potentially change according to the type of value that you select for the limit.

- 1. Positive:** In case the user selects a positive limit for the result, the pattern will be repeated a maximum of `limit - 1` times. Additionally, the length of the final array will not be more than the size of the string itself, and the last entry of the final array will contain the remaining part of the string after the pattern was last matched.
- 2. Negative:** In case the value for the limit entered by the user is negative, the pattern will be repeated as many times as possible. When the value for the limit is set as a negative integer, there will also be no limits on the size of the final resulting array.
- 3. Zero:** When the value of the limit variable is set as zero, the pattern will be repeated as many times as possible. Again, the final resultant array can be of any size. It is, however, important to note that empty strings will be discarded from the final result.

**QUICK  
CHALLENGE**

Write an algorithm which can split any string without using the `split()` method. Print timestamp before and after the code to verify which method is faster.

The following is an example to help you understand how the limit works for the `split()` method:

```
package java11.fundamentals.chapter16;
public class StringSplitWithLimitExample {
    public static void main(String args[]) {
        String myStr = "I@love@java";
        String[] arrOfStr = myStr.split("@", 2);
        for (String piece : arrOfStr) {
            System.out.println(piece);
        }
    }
}
```

The output of the block of code written above will be as follows.



I  
love@java

Since the limit was set as 2 and the string was to split at “@”, there are only two substrings in the result, which is seen in the output.

Similarly, there can also be multiple different characters that can be entered as the parameter at which the string will be split. Each of these characters will have to be explicitly mentioned when they are entered as parameters. The following is an example of how this works:

```
package java11.fundamentals.chapter16;
public class StringSplitOnMultipleCharactersExample {
    public static void main(String args[]) {
        String myStr = "My, Favourite @Programming?Language.Java";
        String[] arrOfStr = myStr.split("[, ?.@]+");
        for (String piece : arrOfStr) {
            System.out.println(piece);
        }
    }
}
```

The result of the block of code written above will be as follows.



My  
Favourite  
@Programming?  
Language  
Java

Since no limits were set for the maximum number of results, the program showed 5 different results, each of which were separated by one of the characters mentioned in the regular expression.

All of the examples mentioned above show that by playing around with your regular expressions and the limits that you set for the number of results that you need, you can modify the types of results that you will get. Moreover, you can also decide the number of substrings that your input string will be divided into.

As mentioned at the beginning of the chapter, strings in Java language are immutable, which means that their contents cannot be changed or modified. This is the reason why new strings have to be created whenever an operation needs to be performed on any string. Fortunately for programmers and application developers of Java language, mutable strings have also been accommodated. Different options are now available for the manipulation of strings without burdening the machine too much, or producing excessive amounts of garbage.

Thanks to the `StringBuilder` and `StringBuffer` classes, manipulation of string objects in Java is far easier than it would be if only the `String` class existed. The `StringBuffer` and `StringBuilder` classes allow strings to easily be manipulated without the need for additional strings to be created. This way, you not only save on garbage that can affect the efficiency of your program, but the performance of your application, program, or block of code also gets much better than it was.

Since both `StringBuilder` and `StringBuffer` are mutable objects in Java, they offer multiple different manipulation options for the strings that are created. Some of the methods offered by these classes include the `insert()`, `delete()`, and `append()` methods that are commonly used for the manipulation of strings. While both the `StringBuffer` and `StringBuilder` classes are essentially used for the manipulation of strings that are created in Java, there are certain significant differences between the two. These will be discussed in detail in following section.

## 16.4 | **StringBuilder and StringBuffer Explained**



The primary reason why both `StringBuilder` and `StringBuffer` are used is for the manipulation of strings in Java language, which makes both of these classes mutable. These are unlike the more popular `String` class, which is used for the creation of strings in Java language. However, this is perhaps the only thing that both the `StringBuilder` and `StringBuffer` classes have in common.

Unlike the `StringBuilder` class, the `StringBuffer` class is thread safe. This means that this class accommodates and modifies data structures only in a way in which they are guaranteed to be executed safely by multiple threads simultaneously. Since the Java language supports the idea of concurrency, this feature of the `StringBuffer` class is of particular interest to developers and programmers who try to incorporate concurrency in their programs and applications. The `StringBuffer` class also contains the `insert()` and `append()` methods, which are popular among programmers who want to manipulate strings in multi-thread environments. Since a wide variety of string operations in Java occur in single-thread environments, the `StringBuilder` class was created without the thread safety option.

The fact that the `StringBuilder` class does not support concurrency or work in multi-threading environments is also the major reason why the `StringBuilder` class is considerably faster than the `StringBuffer` class. Also, the “+” operator that is used for concatenation of strings in Java also uses either the `StringBuilder` class or the `StringBuffer` class internally to perform efficient addition or combination of two or more strings in any application, program, or block of code. Moreover, since `StringBuilder` was introduced in a later version of Java, most of the problems and shortcomings of the `StringBuffer` class have been overcome in the `StringBuilder` class.

### QUICK CHALLENGE

Give a scenario where you would consider `StringBuilder` over `StringBuffer`.

Additionally, since the `StringBuilder` class does not support the idea of synchronization, its performance, efficiency, and speed are considerably different compared to the `StringBuffer` class. Synchronization does not only affect processing power negatively, but also accounts for additional overhead that is completely useless.

To validate the claims made in this section that compares the `StringBuffer` and `StringBuilder` classes, let us see the example of a program that repeatedly performs the `insert()` and `append()` methods on objects of both of these classes. With the help of this program and different test values, we will show how the performance of both of the classes differs and to what extent.

```

package java11.fundamentals.chapter16;
import java.util.GregorianCalendar;
public class StringBufferVsStringBuilderExample {
    public static void main(String[] args) {
        System.gc();
        StringBuffer myStrBuff = new StringBuffer();
        StringBuilder myStrBuild = new StringBuilder();
        runStringBuilder(myStrBuild);
        // Request Garbage Collection to clear the memory
        System.gc();
        runStringBuffer(myStrBuff);
    }

    private static void runStringBuilder(StringBuilder myStr) {
        long begin = new GregorianCalendar().getTimeInMillis();
        long initiateMemory = Runtime.getRuntime().freeMemory();
        for (int j = 0; j < 50000; j++) {
            myStr.append(": " + j);
            myStr.insert(j, "Hello");
        }
        long finish = new GregorianCalendar().getTimeInMillis();
        long stopMemory = Runtime.getRuntime().freeMemory();
        System.out.println("Time Taken for String Builder Append Insert:" + (finish - begin));
        System.out.println("Memory used String Builder Append Insert:" + (initiateMemory - stopMemory));
    }

    private static void runStringBuffer(StringBuffer myStr) {
        long begin = new GregorianCalendar().getTimeInMillis();
        long initiateMemory = Runtime.getRuntime().freeMemory();
        for (int j = 0; j < 50000; j++) {
            myStr.append(": " + j);
            myStr.insert(j, "Hello");
        }
        long finish = new GregorianCalendar().getTimeInMillis();
        long stopMemory = Runtime.getRuntime().freeMemory();
        System.out.println("Time Taken for String Buffer Append Insert:" + (finish - begin));
        System.out.println("Memory used String Buffer Append Insert:" + (initiateMemory - stopMemory));
    }
}

```

The above program produces the following result.

```

Time Taken for String Builder Append Insert:333
Memory used String Builder Append Insert:1603872
Time Taken for String Buffer Append Insert:318
Memory used String Buffer Append Insert:1175040

```

As mentioned above, the value of all variables was changed multiple times, and the program was repeated with each modification to check how the results varied. The value of  $j$  was changed from 1000 to 50,000, and the program was repeated multiple times for both `StringBuilder` and `StringBuffer`. However, there was not too great a difference in the performance of both these classes.

Since the `insert()` method requires a lot of memory and produces plenty of garbage, the same test can also be conducted on both the `StringBuilder` and `StringBuffer` classes without this method to get a better understanding of how the efficiency and performance of both these classes differ.

Additionally, since larger numbers of repetitions will give users and readers a better understanding of how much the results differ. The following is another code which is repeated 50,000,000 times to put things into perspective.

```
package java11.fundamentals.chapter16;
import java.util.GregorianCalendar;
public class StringBufferVsStringBuilderWithoutInsertExample {
    public static void main(String[] args) {
        System.gc();

        StringBuffer myStrBuff = new StringBuffer();
        StringBuilder myStrBuild = new StringBuilder();

        runStringBuilder(myStrBuild);

        // Request Garbage Collection to clear the memory
        System.gc();
        runStringBuffer(myStrBuff);
    }

    private static void runStringBuilder(StringBuilder myStr) {
        long begin = new GregorianCalendar().getTimeInMillis();
        long initiateMemory = Runtime.getRuntime().freeMemory();
        for (int j = 0; j < 50000000; j++) {
            myStr.append(": " + j);
        }
        long finish = new GregorianCalendar().getTimeInMillis();
        long stopMemory = Runtime.getRuntime().freeMemory();
        System.out.println("Time Taken for String Builder Append:" + (finish - begin));
        System.out.println("Memory used String Builder Append:" + (initiateMemory -
stopMemory));
    }

    private static void runStringBuffer(StringBuffer myStr) {
        long begin = new GregorianCalendar().getTimeInMillis();
        long initiateMemory = Runtime.getRuntime().freeMemory();
        for (int j = 0; j < 50000000; j++) {
            myStr.append(": " + j);
        }
        long finish = new GregorianCalendar().getTimeInMillis();
        long stopMemory = Runtime.getRuntime().freeMemory();
        System.out.println("Time Taken for String Buffer Append:" + (finish - begin));
        System.out.println("Memory used String Buffer Append:" + (initiateMemory -
stopMemory));
    }
}
```

The above program produces the following result.

Time Taken for String Builder Append:2200
Memory used String Builder Append:-533019072
Time Taken for String Buffer Append:2401
Memory used String Buffer Append:-14155776

When the test was repeated without the `insert()` method as mentioned earlier, a considerable difference was seen in the amount of time that was taken for the execution of the program to be completed by both the `StringBuilder` and `StringBuffer` classes. With the help of this updated example, it is evident that the `StringBuilder` class can perform better than the `StringBuffer` class even when multi-threading is not used.

Now that you know how string operations work in Java and how the `String`, `StringBuilder`, and `StringBuffer` classes differ from each other, we look into other significant areas of the Java programming language including file operations and I/O operations.

In the sections that follow, we will talk extensively about Java File Class, and how methods and operations can be performed for optimal results and maximum efficiency.

## 16.5 | Java I/O



Ever wondered why you have to write “`import java.io`” at the beginning of most, if not all, of your Java codes? We will answer this question by first understanding Java I/O. As the name suggests, Java I/O is what is used at the back end to process all input and output operations to make your programs work seamlessly and efficiently.

All this is done with the help of streams! Streams are used in Java language to not only expedite the input and output process, but to also make all operations and processing seamless in order to provide optimal results. Additionally, the `java.io` package comprises multiple classes, each of which can prove to be beneficial for a number of different reasons, helping you complete all operations efficiently and effectively. The console for Java comes with three built-in byte streams to make processing easier. These streams are:

1. **System.out:** This is the standard output stream that is used for operations in Java.
2. **System.in:** This is the standard input stream that is used for operations in Java.
3. **System.err:** This is the standard error stream that is used for input output operations in Java.

The following are a few sample codes to show you how each of these streams work to make the input and output process in Java more efficient:

```
package java11.fundamentals.chapter16;
import java.util.Scanner;
public class InputOutputProcessExample {
    public static void main(String args[]) {
        // Following code will create scannerObj object of Scanner class
        Scanner scannerObj = new Scanner(System.in);
        System.out.println("Enter the name of the student");
        // Below line of code ensures that data will be input as string by default
        String studentNAME = scannerObj.next();
        System.out.println("Enter the roll number of the student");
        // Below line of code ensures that data will be input as int by default
        int studentRollNumber = scannerObj.nextInt();
        System.out.println("Enter the marks that the student obtained");
        // Below line of code ensures that data will be input as float by default
        float studentMarks = scannerObj.nextFloat();
        System.out.println("-----Student Report Card-----");
        System.out.println("Student Name:" + studentNAME);
        System.out.println("Student Roll No.:" + studentRollNumber);
        System.out.println("Student Marks:" + studentMarks);
        // Following code is needed to avoid resource leak
        scannerObj.close();
    }
}
```

The above program produces the following result.

```

Enter the name of the student
Mayur
Enter the roll number of the student
17
Enter the marks that the student obtained
100
-----Student Report Card-----
Student Name:Mayur
Student Roll No.:17
Student Marks:100.0

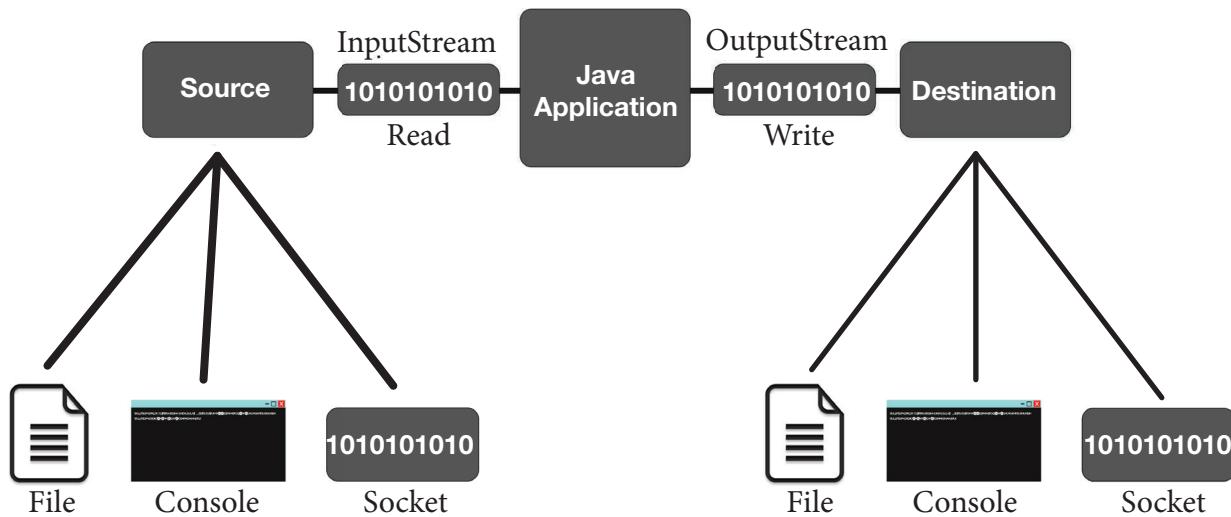
```

After each line of code where an input is requested from the user, the user will be allowed to enter a value according to the data type that is set as default.

To help put things in perspective, here is a line of code that explains how the System.err stream can be used:

```
System.err.println("This is an error message");
```

Now that you understand how the System.in, System.out, and System.err streams can be used, we will discuss the OutputStream and InputStream classes, their most popular methods, and how they differ. See Figure 16.2 to understand the role of InputStream and OutputStream in a Java application.



**Figure 16.2** Role of InputStream and OutputStream.

### 16.5.1 InputStream

One of the major classes of the java.io package is the InputStream class. It provides users with a mechanism to help data be input into Java programs and be read without any problems. An abstract super class by definition, the InputStream class provides programmers and developers with the right tools to not only read bytes of data in singular and array form, but to do so selectively with data streams, mark locations within data streams, determine the number of bytes that are available to be read, and reset the current position within a stream of data.

Here, it is important to note that input streams in Java are opened as soon as they are created. The benefit with this feature is that you do not need to explicitly call the input stream whenever it is needed, and the console will take care of that bit automatically. When the input stream is no longer needed and all information that was requested from the user has already been entered, the `close()` method can be used to close the stream explicitly, or wait for the garbage collection process to be completed, which will automatically close and remove the input stream once it is no longer in use and no longer being referenced.

### 16.5.2 OutputStream

This abstract super class of Java language is essentially used to deal with all outputs in an efficient and effective manner. In addition to providing users with the mechanism and tools to write bytes of data and arrays of bytes, the OutputStream class acts as an interface that is used by multiple other classes to make processing of data more convenient and hassle-free. Much like input streams, output streams can also be closed explicitly by using the `close()` method, or by garbage collection.

Now that you know how both classes differ from each other, we will discuss some of the most popular methods of each of these classes to help you get a better understanding of how they can be used.

### 16.5.3 Methods of OutputStream

There are several different methods of the OutputStream class, each of which serves a unique purpose and offers a unique set of benefits. We will discuss a few of these methods and explain how they differ from each other.

1. The public void `write(int)` method is the method of the OutputStream class that is used to write a single byte to the output stream that is currently in use. It should be noted that this method cannot be used for arrays of bytes.
2. The public void `write(int[])` method is used for writing an array of bytes to the output stream that is currently in use. The difference between the public void `write(int)` and public void `write(int[])` is also evident from the parameters that each of these methods allow. Since the second method has int[] or integer arrays as a parameter, it goes without saying that this method should be used to write an array of bytes to the output stream that is currently in use.
3. The third method of the OutputStream class is the `flush()` method. As the name suggests, this method flushes the output stream that is being used at any given point of time.
4. The last method, `close()`, is used to close the output stream that is currently being used, removing all references and making it eligible for garbage collection.

### 16.5.4 Methods of InputStream

The following methods are part of the InputStream class and can be used to input data to be used by the program or application.

1. The first method is the public abstract int `read()` method. It is used to read the next available byte of data in the input stream. Whenever the method reaches the end of a file or stream, it returns -1 as an indication that there is no longer anything that can be read.
2. The second method is the public int `available()` method. It shows the user an approximate number of bytes that can still be read from the current input stream.
3. The third method is the public void `close()` method of the InputStream class works in the same way as the method of the same name in the OutputStream class, and is used to close the current input stream and make it eligible for garbage collection.



What permissions do you need on a server to use the File Management functionality?

## 16.6 | File Management in Java

File handling or file management is another area of Java that tends to interest a number of people, particularly due to the variety of operations that can be performed on files in the language. We will share some insights into file management in Java programming language.

The FileReader and FileWriter classes are of immense importance when it comes to file management in Java language. Inheriting the OutputStream class, the FileWriter class is used for the creation of files by writing characters. While certain assumptions are made by the constructors of this class, programmers and developers of Java are free to specify values by themselves by creating their own constructors. Some of the constructors of the FileWriter class are as follows:

1. **`FileWriter(File, File)`:** As the name suggests, this constructor creates a FileWriter object when a File object is input as a parameter.
2. **`FileWriter(String filename)`:** Since the parameter for this constructor is a String, the constructor will create a FileWriter object whenever a file name is input as a parameter.



Here are some of the methods that are available through the `FileWriter` class:

1. `public void write (int c)`: This method is used to write a single character into the stream that is being created.
2. `public void write( char[] stir)`: The character array that needs to be inserted to the output stream will be input into this method as a parameter.

Inheriting the `InputStreamReader` class, the `FileReader` class is used to read data from any file – one character at a time. It is important to note that the `FileReader` class can only be used to read data in the form of characters, while the `FileInputStream` class is used to read data in the form of raw bytes. Here are some of the constructors of the `FileReader` class:

1. `FileReader(File,File)`: As evident from the parameters of this constructor, a `FileReader` object is created when a `File` object is inserted as a parameter.
2. `FileReader(String filename)`: This constructor creates a `FileReader` object given that the name of the file that needs to be read from is input as a parameter.

Here are some of the methods that are available through the `FileReader` class:

1. `public int read()`: This method is used to read a single character from the stream that is available.
2. `public int read(char[] cbuff)`: This method is used to read characters into an array.



Can you read a file from a remote server?

## Summary

With this, we have completed the chapter on Strings and how they work in Java language. We have also described in detail about input and output in Java programming language, and the methods and operations that are available to make processing more efficient and optimized.

In this chapter, we have learned the following concepts:

1. String operations and its usage.
2. Various string functions such as concatenation and split.
3. Differences between `StringBuffer` and `StringBuilder`.
4. Various I/O operations.
5. File management with file reader and file writer.

In Chapter 17, we will learn about data structures and its types, such as primitive and non-primitive data structures, and how to use them in a program.

## Multiple-Choice Questions

1. Which of the following sentences is false about `String` in Java?
  - (a) We can extend `String` class like `StringBuffer` does it.
  - (b) `String` class is defined in `java.util` package.
  - (c) `String` is immutable in Java.
  - (d) `String` is thread-safe in Java.
2. Which of the following methods of `String` class can be utilized for testing strings for equality?
  - (a) `isequal()`
  - (b) `isequals()`
  - (c) `equal()`
  - (d) `equals()`
3. \_\_\_\_\_ class is used for reading characters in a file.
  - (a) `FileWriter`
  - (b) `FileReader`
  - (c) `FileInputStream`
  - (d) `InputStreamReader`
4. \_\_\_\_\_ method is used for reading characters in a file.
  - (a) `read()`
  - (b) `scan()`
  - (c) `get()`
  - (d) `readFileInput()`

5. \_\_\_\_\_ method is used for writing into a file.
- (a) putfile()  
 (b) write()  
 (c) writefile()  
 (d) put()

## Review Questions

---

1. What are various operations you can perform on String?
2. Is String immutable?
3. How to read and write content into a file?
4. How do we use StringBuilder?
5. What are the advantages of using StringBuilder over StringBuffer?
6. How do we use InputStream? Give one example.
7. How do we use OutputStream? Give one example.

## Exercises

---

1. Write a program to test the difference between StringBuffer and StringBuilder.
2. Write a program to test all the string manipulations using various functions.
3. Create a program which writes content into a file and read from the file.

## Project Idea

---

Create a job application portal that can accept an applicant's CV and process its data. The program should be able to

read the CV file and collect the applicant's name, address, education, employment details and show it on a page.

## Recommended Readings

---

1. Oracle Tutorials: Regular Expressions – [https://www.w3schools.com/java/java\\_strings.asp](https://www.w3schools.com/java/java_strings.asp)
2. W3Schools – <https://docs.oracle.com/javase/tutorial/essential/regex/>
3. Oracle Tutorials: Manipulating Characters in a String – <https://docs.oracle.com/javase/tutorial/java/data/manipstrings.html>
4. Oracle Tutorials: Basic I/O – <https://docs.oracle.com/javase/tutorial/essential/io/>

# Data Structure and Integration in Program

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Various data structures.
- Difference between primitive and non-primitive data structures.
- Difference between tree and graph.
- Various tree data structures.
- Binary tree and how it is useful.
- Best data structure for your needs.

### 17.1 | Introduction

Data structures are essentially arrangements that can be used for storage and manipulation of the internal data of a computer program. What is interesting to note here is that most novice developers and programmers of Java programming language start using data structures long before they even know what they really are or how they are different from other data types.

While there are a lot of different data structures, some of the most commonly used ones include stacks linked lists and arrays. Having a strong understanding of the different types of data structures available in Java language and how they differ from each other is imperative for a number of reasons. The choice of data structure does not only affect the time taken by the application to perform crucial tasks; however, the choice of data structure has a strong connection with the effort that is required for implementation and the performance of the block of code, program, or application.

Since you now understand why it is important to know the difference between each of the data structures that are available, we will discuss a number of different data structures in great detail, take a look at their advantages and disadvantages, and explain how each of them can be used in programs.

### 17.2 | Introduction to Data Structures



Data structures are closely related to abstract data types (ADTs), which is another extremely important area in Java programming language. Best termed as a unique mathematical model for data types, an ADT is essentially defined by the user based on how it is expected to act in the program, block of code, or application where it is to be used, and the operations that are likely to be performed on it.

While data structures are essentially based on ADTs, the difference lies in the fact that there is a concrete implementation mechanism in place for data structures, and they cannot be arbitrarily used like ADTs.

As complex and beneficial as they are for efficient programming that gives optimal results, data structures can be classified and categorized into a number of different groups, each of which has its own characteristics, features, and properties. The focus of this chapter will be exclusively on data structures, their types, and their correct usage.

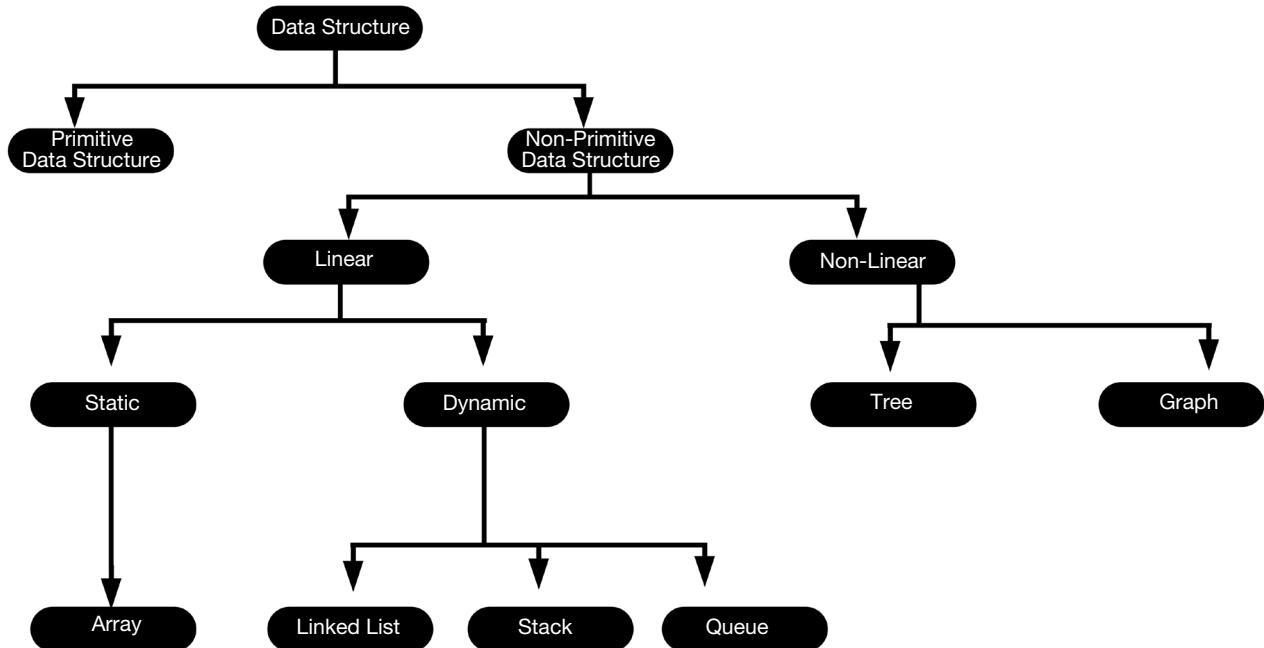


How do you program without any data structure?



## 17.3 | Classification of Data Structures

As mentioned, there are a number of different ways in which data structures of Java language can be classified. The most common classifications are primitive data structures and non-primitive data structures. Figure 17.1 shows the data structure hierarchy.



**Figure 17.1** Data structure hierarchy.

### 17.3.1 Primitive Data Structures

Primitive data structures are nothing but the basic data types that are available in every programming language. They are used for the most fundamental of reasons. These predefined ways of storing data in Java language come with a limited set of operations that can be performed on them, essentially setting a limit to the number and types of situations for which they can prove to be beneficial.

We have learned about this in the Chapter 11, but let us revise it again. There are eight different primitive data structures available in Java language, namely, int, char, float, Boolean, long, double, short, and byte. Often referred to as the fundamental building blocks for the manipulation of data in Java, these data structures can be used to store the simplest types of values – that too of a single kind. Since primitive data types are very limited in what they offer, it is no surprise that there are certain situations where primitive data structures alone do not suffice. And that is where derived data structures come into the picture.

Derived data structures and user-defined data structures not only offer a much wider range of applications, but their benefits are also unique in a number of ways. We will learn about non-primitive data structures and how they differ from their primitive counterparts in the following sections. For now, we will define each of the primitive data structure of Java language along with examples of how primitive data structures can be declared and used in any program, application, or block of code, and what they can be used to achieve.

- Byte:** The Byte primitive is one of the basic predefined data structures of Java language, which is found in the form of an 8-bit signed two's complement integer. The Byte primitive has a maximum value of 127, and can take values as low as -128. The default value of this data structure is set as 0.

The fact that the Byte primitive requires far less storage space than the int type is one of the major reasons why it is so popular among developers. A Byte is four times smaller than an integer, which is why it can be easily used in certain scenarios in place of integers, especially when storage space is a concern.

Here is an example of how a Byte primitive can be declared:

```
byte a = -37;
```

2. **Short:** Like Byte, the Short primitive is also a two's complement signed integer. The difference between the two, however, lies in the fact that the Short data structure has 16-bit values. What this means is that the range of values for Short is much wider than for Byte, making the maximum possible value for Short 32,767 and the minimum possible value -32,768.

Like the Byte primitive, the Short primitive also has a default value of 0, and is used especially in situations where storage is a concern. This is because the Short type takes two times less space than the integer type.

Here is an example of how the Short primitive can be declared to be used in any program written using Java language:

```
short s1 = 32000;
```

3. **Int:** A signed two's complement integer, the Int primitive allows a 32-bit value, making the range of the data structure much larger than both the Short and Byte primitives combined. Since the primitive allows 32-bit values, the maximum value that can be stored in an Int type is  $2^{31} - 1$ , or 2,147,483,647, whereas the minimum value that is allowed in the Int type is  $-2^{31}$ , or -2,147,483,648. The default value for the Int type is 0.

Even though the Int type takes up a lot more memory than the Short and Byte types, it is often the data structure of choice for integer values, unless memory or storage space is a concern.

Here is an example of how an Int type variable can be declared in Java language:

```
int a = 500000;
int b = -500000;
```

4. **Long:** The Long primitive of Java language is a signed two's complement integer that can have a 64-bit value. Needless to say, the range of values allowed by Long is far wider than all other primitives in Java, with possible values ranging from a minimum of  $-2^{63}$ , or -9,223,372,036,854,775,808, to a maximum of  $2^{63} - 1$ , or 9,223,372,036,854,775,807.

Since this primitive takes up more memory and requires more storage space than variables of the Int type, the Long primitive is only used for integer values that will not be possible with the Int primitive. Additionally, values of the Long primitive are easily differentiable from values of variables of other primitives because they are always terminated with L. Similar to the default values of other variables discussed above, the default value for the Long primitive is 0L.

Here is an example of how a variable of the Long type can be declared in Java language:

```
long a = 19823290832L;
```

5. **Float:** The Float data structure in Java language is an extremely interesting and useful one for many reasons. The Float primitive allows single precision 32-bit values, and is primarily used for saving memory in situations where arrays of floating point numbers need to be used. Like the values of the Long type, values of the Float primitive are also easily differentiable thanks to the "f" at the end of all values of the data type.

The default value of the Float primitive is 0.0f, and the data type is never used for situations in which precise values are needed.

Here is an example of how a variable of the float type can be declared in Java language:

```
float f1 = 2.5f;
```

- 6. Double:** As the name suggests, the Double primitive is a double precision data structure in Java language that supports 64-bit values, giving it a wider range of possible values than the float data structure. This primitive is generally used as the default choice of programmers when they need to deal with decimal values.

Like the Float data structure, the Double primitive should never be used in situations where accuracy and precise values are imperative for the integrity of the program or application to be maintained. Additionally, values of Double variables are also easily differentiable as they end with a “d”.

Here is an example of how a variable of the Double primitive can be declared in Java language:

```
double d1 = 234.5;
```

- 7. Boolean:** The Boolean primitive is used for representation of a single bit of data in Java language. Unlike the other primitives and the range of values that were allowed in them as discussed above, the Boolean primitive allows only two values – false and true.

The only area where this primitive can be used is for tracking whether a condition will be true or false. The default value for variables of the Boolean type is set as false.

Here is an example of how a variable of the Boolean primitive can be declared in Java language:

```
boolean var = true;
```

- 8. Char:** The Char primitive is used for representation of a single 16-bit Unicode character in Java language. The range of values allowed by the Char primitive is rather limited, with the minimum possible value as 0, or “\u0000”, and the maximum possible value 65,535, or “\uffff”.

The Char primitive is only used in situations where a single character needs to be stored.

Here is an example of how a variable of the Char primitive can be declared in Java language:

```
char c1 = 'b';
```

As evident from the explanation of each of the primitives and the range of values permissible by each of them, it goes without saying that another mechanism for the storage of variables and their manipulation was crucial. Primitives of Java language are not only limited in the operations that can be performed on them, but the range of values allowed by them also limits the possibilities.

And this is exactly why non-primitive data structures such as arrays, stacks, and linked lists exist.

While they are essentially made up of primitive data structures, non-primitive data structures provide a lot more room for operations, and leave users with more to play around.

In the following section, we will learn about the most common non-primitive data types and explain how each of them can be used, their advantages, and how they differ from each other.

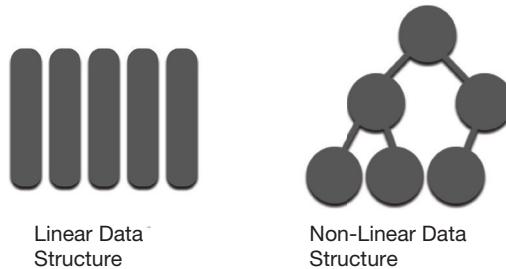


What is the difference between Float and Double?

### 17.3.2 Non-Primitive Data Structures

While non-primitive data structures are based on their primitive counterparts, the range of functionality offered by non-primitive data structures and the benefits that they offer are greater than primitive data structures. As shown in Figure 17.2, non-primitive data structures can be classified into two major groups – linear non-primitive data structures

and non-linear non-primitive data structures. Table 17.1 lists the differences between linear and non-linear non-primitive data structures.



**Figure 17.2** Representation of linear and non-linear data structures.

**Table 17.1** Linear data structure vs non-linear data structure

Parameter	Linear Data Structure	Non-Linear Data Structure
<b>Basic</b>	Elements are arranged adjacent to each other	Elements are arranged in a sorted order
<b>Traversing of the data</b>	Data elements are traversed in one go	Data elements cannot be traversed in one go
<b>Ease of implementation</b>	Simple	Complex
<b>Levels involved</b>	Single	Multiple
<b>Examples</b>	Array, LinkedList, Queue, Stack, etc.	Graph, Tree
<b>Memory utilization</b>	Inefficient	Efficient

#### QUICK CHALLENGE

Define a situation in which you could use non-linear data structure.

### 17.3.2.1 Linear Non-Primitive Data Structures

For a data structure to be considered as linear, it is imperative for the elements that make it should form a linear list. Another requirement is that the constituent elements of the data structure should be connected to each other adjacently and follow a certain order. Additionally, for a data structure to be considered linear, the memory that it consumes should also be in linear fashion, and the storage of elements in the memory must be sequential.

Linear non-primitive are also differentiable from their non-linear counterparts because a certain amount of memory has to be declared in advance for the linear data structure, unlike in the case of non-linear data structures. While this requirement often results in loss of memory and mismanagement due to improper utilization of memory, doing so is imperative if you want to use linear data structures. Additionally, since memory and element storage for linear data structures works in a linear and sequential fashion, constituent elements of a linear structure can only be reached sequentially, and only a single element of the data structure can be visited without a reference. Since it is imperative for adjacency relationships to be maintained for data structures, the operations that can be performed on linear data structures are also limited in a number of ways. What this means is that it is not possible for one to insert or delete constituent elements of a linear data structure arbitrarily, and all operations must be performed in a sequential fashion. Here are some of the operations that can be performed on linear data structures in Java language:

1. Addition of elements.
2. Deletion of elements.
3. Traversing the data structure.

4. Sorting the constituent elements of the data structure.
5. Searching for a constituent of the linear data structure.

Now, we will look at a few of the most popular linear data structures in detail, and learn how they can be used in programs or applications.

### 17.3.2.1.1 Arrays

Arrays are perhaps the most popular data structure in Java and other programming languages. Often built into programming languages, arrays act as a great starting point to not only introduce novice programmers to the concept of data structures, but also to understand how object-oriented programming (OOP) and data structures go hand in hand. In this section, we will learn how to create arrays from scratch, use them effectively and efficiently, and perform operations on them. This will help users get an idea of how data structures really work; we will then move on to more complicated linear and non-linear data structures that are used in Java language.

Arrays in Java are created dynamically and can contain multiple elements that essentially define the size or length of the array. What this means is that if an array contains  $x$  elements, the length of the array will be  $x$ .

While all of the constituent elements of an array have the same name, they each have a unique reference based on where they lie in the linear sequence of the data structure. To reference any particular element in the data structure, you will need the index of the element in the array, which will always be in the form of a non-negative integer. Since the index of the first element is always 0, the last constituent element of any particular array will always be  $x - 1$ , where  $x$  is the length, size, or total number of elements present in the data structure.

It is important to note that even though it was mentioned that an array can contain multiple elements, the condition is that all of these elements must be of the same type, which is often called the component type of an array. What this means is that even if an integer array has been created, you will not be allowed to enter float values in the array. If an integer array needs to be created as in the case mentioned above, we will declare the integer array as follows:

```
int age[]; // This line of code declares an integer array to hold ages
```

It is also important to note that arrays in Java can have multiple dimensions, but that does not change the fact that there are no restrictions on the element type or component type of the array. Additionally, whenever a variable of the array type is created, it does not set aside any amount of memory or create the object for the array. Instead, the only thing that really happens is the creation of the variable for the array – a variable that may be used to contain the reference to the array itself.

Now that you have a basic understanding of how arrays work, their characteristics, and the operations that can be performed on them, we will focus on the practical implementation of arrays and how that can be done in Java.

**Declaration of simple arrays:** An array in Java language is essentially a list of elements that have the same type and are referred to by the same name. As seen above, there is not much that needs to be done to declare an array in Java besides adding a pair of square brackets after the name of the array that you wish to create.

Another way to do this is by adding the pair of square brackets after the type of array, as follows:

```
float[ ] temperature;
```

Therefore, the permissible syntax for the declaration of an array is as follows:

```
type[ ] name;
```

or

```
type name[ ];
```

As mentioned above, simply declaring an array does not suffice. Instead, you will have to create or initialize your array for you to be able to effectively use it in the program, application, or block of code. Next, we will explain how an array can be created, and how memory can be allocated to ensure that the array that was declared is functional and can be used in the program.

**Creating a one-dimensional array:** Whenever an array is to be created to be used in Java, the **new** keyword will be used, and the length or size of the array will be specified in square brackets. Here is an example of how this will work:

```
Int age[ ]; // this line of code declares a one-dimensional array of the integer type named age
age = new int[4]; // this line of code creates a new array after declaring it
```

**Initialization of a one-dimensional array:** This is perhaps the easiest and simplest part of the process. The only thing that needs to be done for initialization of an array in Java is to place values in curly braces separated by commas. Here is how this can be done:

```
int age[4] = {11,22,35,64};
```

Now that you understand how an array can be declared, created, and initialized in Java, here is a sample code that will help put things into perspective:

```
package java11.fundamentals.chapter17;
public class ArrayExample {
    public static void main(String args[]) {
        // Following code declares an int array
        int myIntArr[] = { 1, 5, 993, 35 };
        // Following code iterates through array elements and print them one by one
        for (int b = 0; b < myIntArr.length; b++) {
            System.out.println(myIntArr[b]);
        }
    }
}
```

The above program produces the following result.

1
5
993
35

As seen above, the declaration, creation, and initialization of arrays in Java language is extremely easy and simple. However, there are advantages and disadvantages to the concept of arrays in Java language.

Although arrays make for a great data structure and provide a resourceful mechanism for the storage and manipulation of large amounts of data, the major problem is that they can only be used to store data elements of a single type. Moreover, once an array has been declared, it is not possible for it to be changed in any way. What this means is that once you have declared an array, it will not be possible to increase or decrease its size, resulting in memory wastage.

**QUICK  
CHALLENGE**

Define two arrays of same size and integer type. Write a program to create a third array which contains the values which are the result of multiplication of each element from the first array and second array. [The first element of first array will get multiplied with the first element of second array, the second element of first array will get multiplied with the second element of second array, and so forth.]

### 17.3.2.1.2 Lists and Queue

The List interface in Java language is particularly interesting for a number of reasons. While there are several implementations of the List interface, they all work on an identical principle. Much like arrays, elements in Lists of Java can also be accessed according to their index or position in the list. The same mechanism is also used for the addition of elements, and duplication is possible within lists as well. Additionally, since List extends Collection in Java language, all of the operations of Collection are supported by List as well.

Queue is another data structure which works on the First In, First Out (FIFO) principal. We have covered these topics in detail in Chapter 13.

### 17.3.2.2 Non-Linear Non-Primitive Data Structures

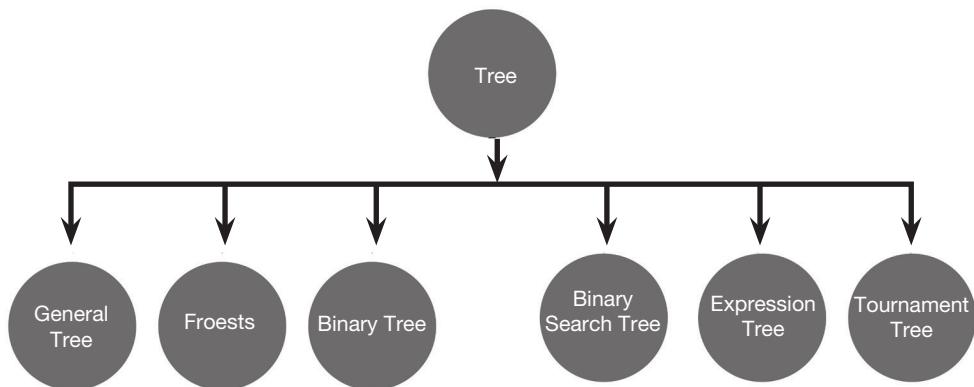
The following data structures come under the non-linear non-primitive data structures category. Data in this structure is not arranged sequentially but in a sorted order. In this type of structure, all the operations are done in a non-sequential manner such as traversal of data elements, insertion, and deletion.

These types of non-linear data structures are memory efficient, as they use memory resourcefully and do not need pre-memory allocation. The two types of data structures available are *tree* and *graph*. Data is arranged in a hierarchical relationship in the tree structure, which involves the relationship between the child, parent, and grandparent.

#### 17.3.2.2.1 Tree Data Structure

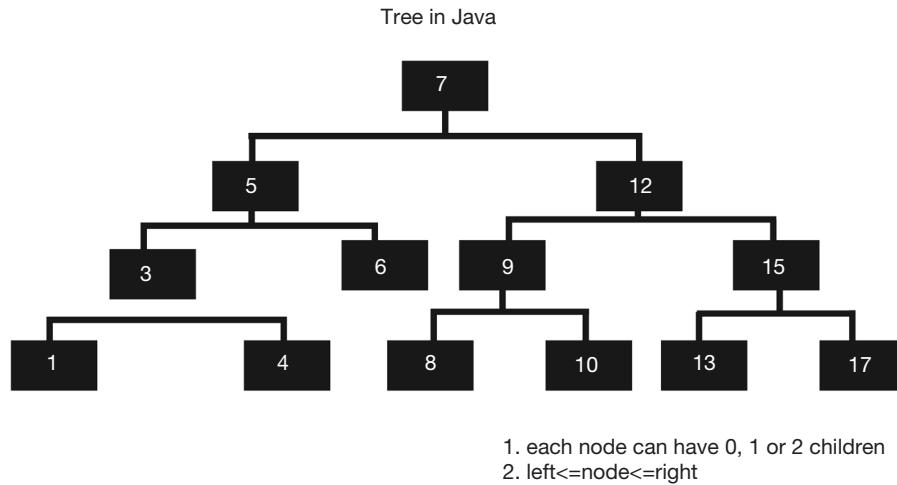
As shown in Figure 17.3, the following are tree related data structures:

1. General tree.
2. Forests.
3. Binary tree.
4. Binary search tree.
5. Expression tree.
6. Tournament tree.



**Figure 17.3** Tree data structures.

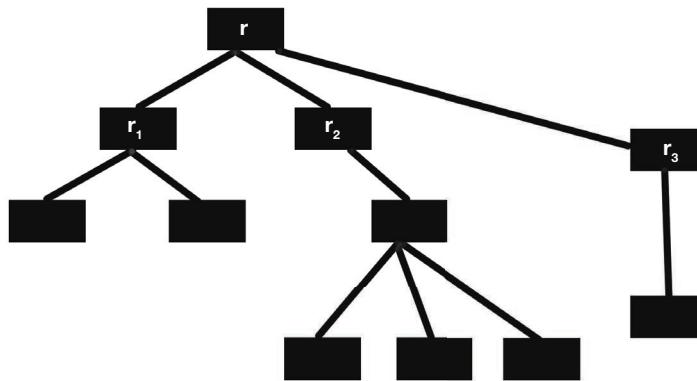
Most of the time, you will be using binary tree and binary search tree data structures. These are two of the popular data structures commonly used to solve many problems. Figure 17.4 shows the tree representation in Java.



**Figure 17.4** Tree representation.

1. **General trees:** Many times we encounter a set of data which is not in a format that can be handled by binary tree. Binary tree can have maximum 2 nodes. Hence, cases like displaying organization hierarchy cannot be handled by binary tree as the CEO (root node) can have more than 2 vice presidents under it. For this type of data, we need a tree data structure which can handle multiple nodes under any root node. This data structure is known as general tree, which is shown in Figure 17.5.

A general tree  $T$  is a tree which has one root node  $r$  and finite set of one or more nodes. Tree  $T$  starts with root node  $r$ . If the first set ( $T \setminus \{r\}$ ) is not empty, then the rest of the nodes are divided into  $n \geq 0$  disjoint subset trees like  $T_1, T_2, \dots, T_n$  which are called *subtrees*. Each of these trees has a root node which looks like  $r_1, r_2, \dots, r_n$ , respectively. These root nodes are children of  $r$ .

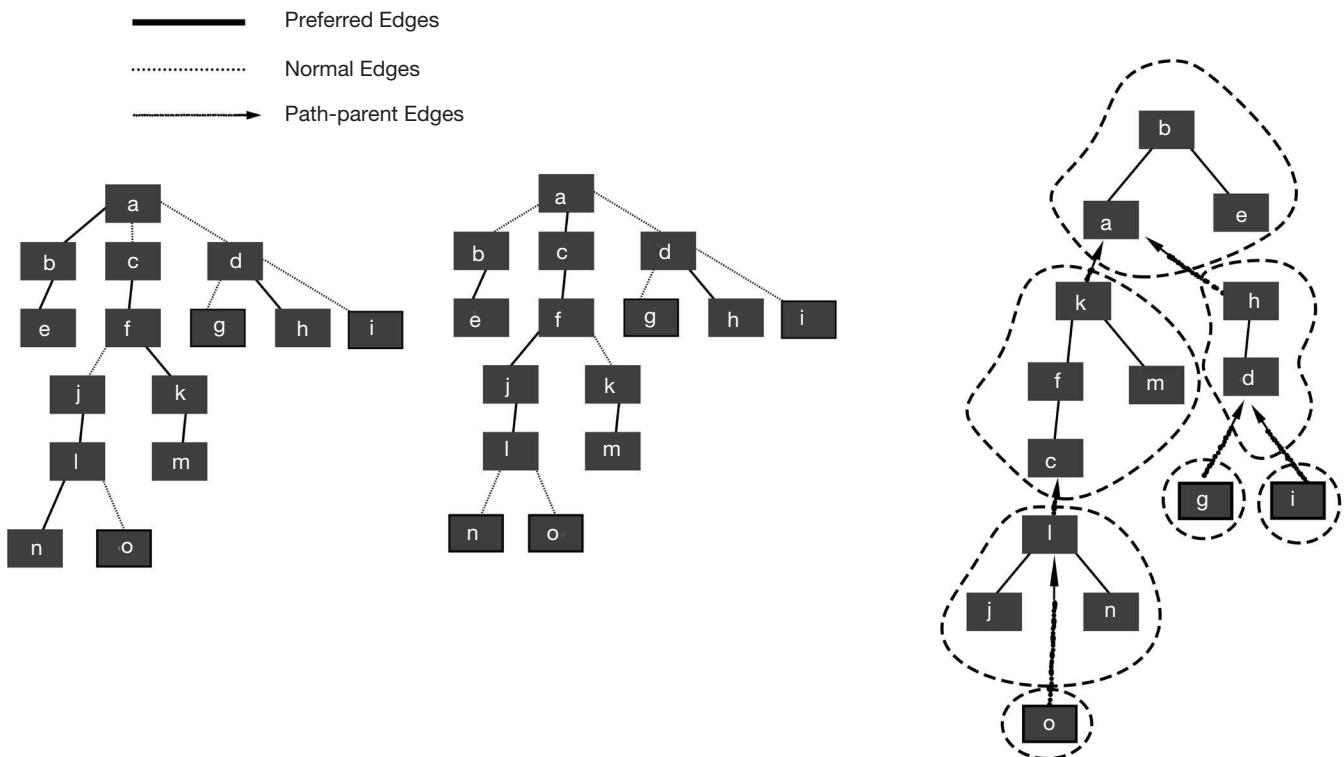


**Figure 17.5** General tree structure.

**QUICK CHALLENGE**

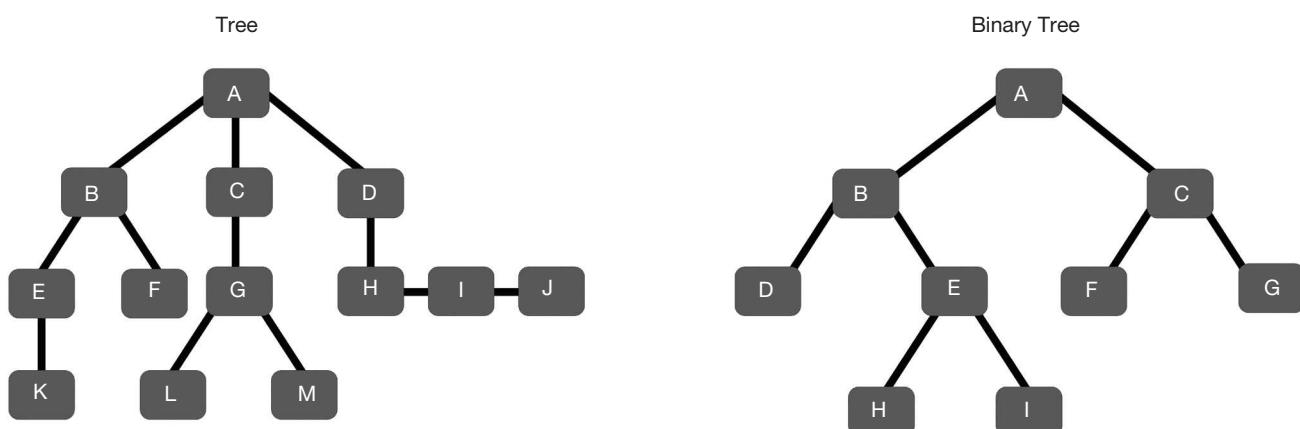
Give an example of a real-life scenario which is suitable to be represented as a general tree.

2. **Forests:** An arbitrary set of trees is called Forest. Figure 17.6 shows representation of Forests. A Forest can start with one root node; it then grows in an ordered fashion, with one node having any number of child nodes. The children of a node are sequenced as first, second, and so on. Contrary to the binary tree, this tree does not have a concept of left and right. However, we can sequentially draw this tree from left to right. An Ordered Forest is nothing but an ordered set of ordered trees, which are also termed as Orchard.



**Figure 17.6** Representation of Forests.

**3. Binary tree:** As we already know, tree data structure is hierarchical in nature. Binary Tree is a subset of Tree in which each node has at most two children. Figure 17.7 shows Tree vs Binary Tree. These children are referred as the left and right child. This tree is implemented using links. The topmost node is the pointer in the tree. If tree is empty, then the value of root is NULL.



**Figure 17.7** Representation of Tree vs Binary Tree.

A node has three parts: (a) data, (b) pointer to left child, and (c) pointer to right child.

There are two ways in which a binary tree can be traversed:

- (a) **Depth-first traversal:** In this case, there are three ways to traverse:
  - In the first way, it starts with left node, then root node, and then right node is accessed. This is called *inorder (Left-Root-Right) traversal*.
  - In the second way, it starts with the root node, then moves to the left node, and then comes to the right node. This type of traversal is called *preorder (Root-Left-Right) traversal*.
  - The third way is to start with left node, then move to the right node, and finally come to the root node. This is called *postorder (Left-Right-Root) traversal*.
- (b) **Breadth-first traversal:** In this case, the traversal takes place level-by-level. As name suggests, the focus is on the breadth of the level. In other words, at every level, the traversal takes place on the full width of the tree before going to the next level. Let us take an example of the binary tree shown in Figure 17.7. In this example, node A is at level 1, nodes B and C at level 2, nodes D, E, F, and G are at level 3, and nodes H and I are at level 4. In this case, the algorithm will first traverse through node A, then through nodes B and C, then nodes D, E, F, G, and finally it will traverse through nodes H and I.

Following are some of the properties of binary tree:

- (a) The maximum number of nodes at level  $x$  of a binary tree is always  $2^{x-1}$ .
- (b) The maximum number of nodes of height  $y$  is  $2^{y-1}$ .
- (c) The maximum number of levels or height in tree with  $n$  nodes is  $\log_2(n+1)$ .
- (d) Every node in the tree has 0 or 2 children.
- (e) Total number of nodes with 2 children are always less by one than the number of leaf nodes (i.e.,  $L = X + 1$ , where  $L$  is number of leaf nodes and  $X$  is internal nodes with two children).



Construct a binary tree for the following array:

```
myArray[] = {1, 2, 3, 4, 5, 6}
```

4. **Binary search tree:** This tree is similar to the Binary Tree. It is mainly used for faster search over LinkedList, but it is slower than arrays. For insertion and deletion, it is better than arrays but not LinkedList. This is an important data structure that you should be aware of. It provides efficient search with  $O(\log n)$  complexity.



Can a Binary Search Tree's topmost node contain a null node?

It has three more other properties than Binary Tree.

- (a) It contains keys less than the node's key on the left side.
- (b) It contains keys greater than the node's key on the right side.
- (c) Both sides, left and right, must be a Binary Search Tree.

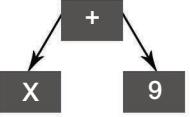
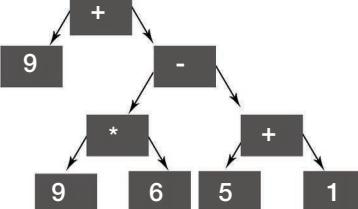
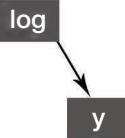
A Binary Search Tree is quite useful in applications such as e-commerce, where products are getting added or deleted from the inventory constantly and then presented in a sorted order.



Can a Binary Search Tree's topmost node contain more than two elements?

5. **Expression tree:** As name suggests, an Expression Tree is one made up of expression, wherein each internal node contains an operator and each leaf node contains an operand. For example, in the expression  $x + 9$ ,  $x$  and  $9$  are operands and  $+$  is the operator. Hence, the internal node will start with  $+$ , and on the left we will have  $x$  and on right we will have  $9$ . Table 17.2 shows a few examples of Expression Trees.

**Table 17.2** Examples of Expression Trees

Expression	Expression Tree	Inorder Traversal Result
$(x+9)$		$x+9$
$9+(9*6-(5+1))$		$9+9*6-5+1$
$\log(y)$		$\log(y)$

6. **Tournament trees:** A complete Binary Tree with  $n$  external nodes and  $n - 1$  internal nodes is called Tournament Tree. All the external nodes are characterized as players and all the internal nodes are characterized as winners. Winner node is situated between the two external nodes termed as players. As we have just discussed, a Tournament Tree gets  $n - 1$  internal nodes and  $n$  external nodes. Thus, to find the winner, we need to eliminate  $n - 1$  players. In simple term comparisons, this means that we need a minimum of  $n - 1$  games.

There are two types of Tournament Trees:

1. **Winner tree:** A Winner Tree can be defined as the complete Binary Tree, where every node characterizes the smaller or greater of its two children. As in the case of the Binary Tree, it starts with the root, so root becomes the representation of the smaller or greater node. The Tournament Tree winner can be found by looking at the smallest or greatest  $n$  key in all the sequences. The time complexity of this tree is  $O(\log n)$ .
2. **Loser tree:** A Loser Tree can be defined as the complete Binary Tree with  $n$  external nodes and  $n - 1$  internal nodes. The loser is stored in the internal nodes of the tree. The root node at [0] is the winner of the tournament. The corresponding node is the loser node.

**Uses of tournament tree:** The following are the uses of this type of tree:

1. Finding the smallest and largest element in the array.
2. Sorting.
3. Merging M-way. (M-way merge is like merging M sorted arrays to get single sorted array).



What are the benefits of using a Tournament Tree?

### 17.3.2.3 Graph Structure

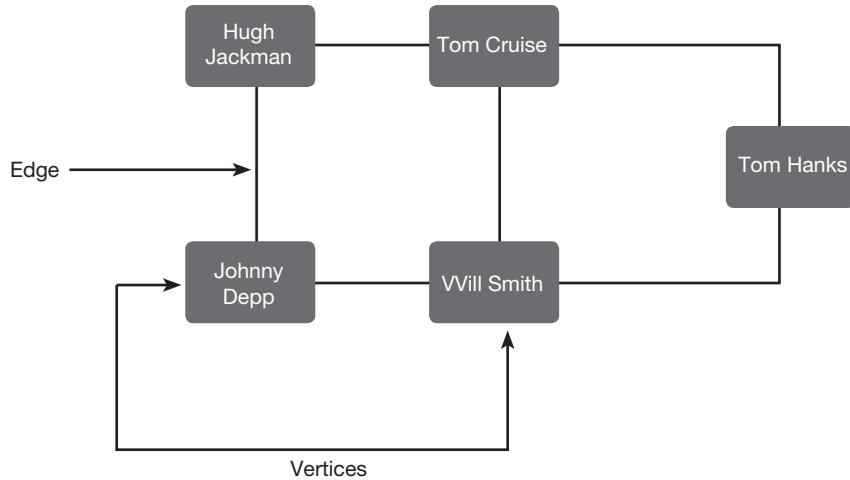
Graph data structure stores connected data where nodes are connected to each other in a network fashion. Think of a social media platform. You are connected to your friend, your friend connects back to you, there is someone connected to your friend, there may be someone whom you also know connected to your friend, and so on and so forth. This is how graph is setup. A

graph contains vertices and edges. The vertex represents an entity like people and the edge is the relationship between those entities.



Can a graph be useful to store a dictionary values?

Figure 17.8 shows a graph, edges, and vertices.

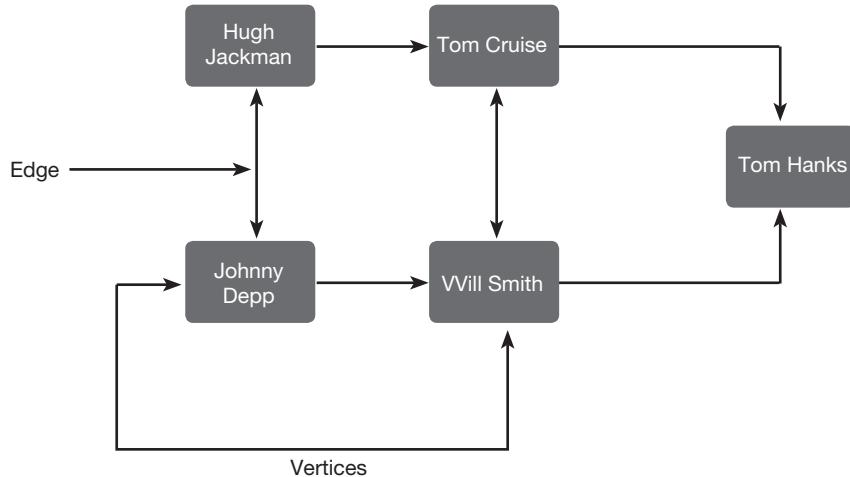


**Figure 17.8** Example of a graph.

In the above example, you can see that the nodes are named as Johnny Depp, Will Smith, Hugh Jackman, Tom Cruise, and Tom Hanks. These are entities and the connection between these nodes are edges.

There are a few variations to the graph data structure of Figure 17.8 . These are discussed in the following subsections.

#### 17.3.2.3.1 Directed Graph



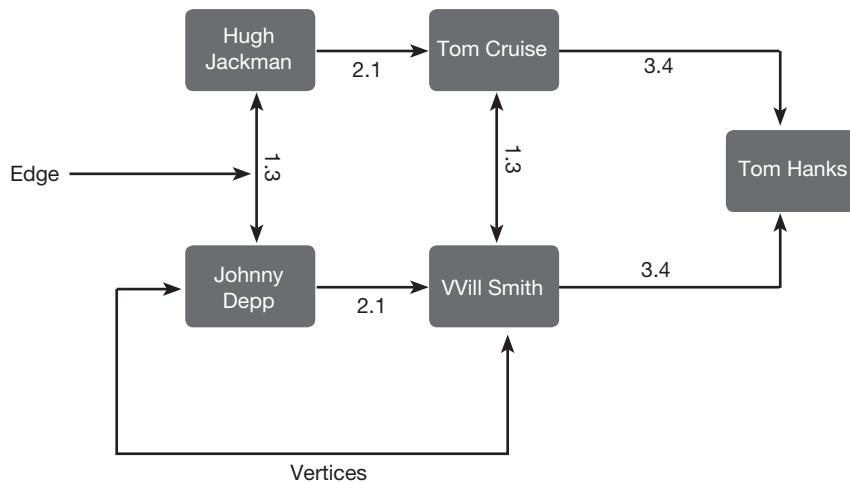
**Figure 17.9** Example of directed graph.

In the above example shown in Figure 17.8, we do not see any direction. That is, the nodes are just connected and do not carry any specific direction. If we add direction to the edges, then the graph becomes a directed graph. It means that the edges

point the relationship direction between nodes. In the social media example, this could be used to show who sends a friendship request. These edges can carry bidirectional relationship as well as shown in 17.9. Bidirectional relationship exists between two nodes which are connected both ways. In other words, the connection between node A and node B is taking place from node A to node B and vice versa.

### 17.3.2.3.2 Weighted Graph

We have now seen a graph with edges and one with direction-based edges. If these edges carry relative weight, they become a weighted graph as shown in Figure 17.10. In the social media example, each connection carries direction, like the connection between you and your friend A. If we add the number of years you know your friend to the edge, then that becomes the weight of the relationship.



**Figure 17.10** Example of weighted graph.

### 17.3.2.4 Graph Representations

A graph has two commonly used representations – adjacency matrix and adjacency list. It also has other representations that are less commonly used – incidence matrix and incidence list.

#### 17.3.2.4.1 Adjacency Matrix

An adjacency matrix is a square matrix. In simple terms, as shown in Figure 17.11, an adjacency matrix shows if the elements in the graph are next to each other. The number of vertices in the graph defines the dimensions of the graph. This matrix has simple representation, which contains values of 0 or 1. A value of 1 indicates that there is an adjacency between row and column. And a value of 0 indicates that there is no adjacency between row and column. Let us take our example and see how it can be represented as a graph.



Can an adjacency matrix be useful to define edge weight?

This is the easiest representation to implement and follow. The complexity of edge removal is  $O(1)$  time and the edge find between two vertices can also be done in  $O(1)$  time. However, the adjacency matrix takes a larger space of  $O(V^2)$  time (here, V is vertex set) even though it may contain lesser number of edges. The edge addition takes  $O(V^2)$  time.

	Hugh Jackman	Tom Cruise	Tom Hanks	Johnny Depp	Will Smith
Hugh Jackman	0	1	0	1	0
Tom Cruise	1	0	1	0	0
Tom Hanks	0	1	0	0	1
Johnny Depp	1	0	0	0	1
Will Smith	0	1	1	1	0

Figure 17.11 Example of adjacency matrix.

#### 17.3.2.4.2 Adjacency List

As shown in Figure 17.12, an adjacency list is simply an array of lists. The number of vertices in the graph determines the size of the array. Array[i] represents the  $i$ th vertex's adjacent vertices. This list can also represent a weighted graph. The weights of edges in the weighted graphs are represented as lists of pairs. The adjacency list representation takes less space  $O(|V|+|E|)$ , where  $V$  is vertex set and  $E$  is edges. In a worst-case scenario, it may take  $O(V^2)$  space, even though there may be  $C(V, 2)$  number of edges. Adding a vertex is relatively easier than in an adjacency matrix. However, the edge find process between two nodes may be expensive, such as  $O(V)$  times.



Figure 17.12 Example of adjacency list.

### 17.3.2.4.3 Incidence Matrix

Incidence matrix is useful to show the relationship between objects of two classes. This matrix is built in a way that it contains one row for each element of first class and one column for each element of second class. If first class row  $x$  and second class row  $y$  are related, it contains 1 in row  $x$  and column  $y$  which is called incident in this context and it contains 0 if they are not related.

Incidence matrices are frequently used in graph structure, such as undirected and directed graphs.

- Undirected graph:** Let us take the example shown in Figure 17.13 of an undirected graph,  $X$ . This graph contains vertices and edges; say,  $n$  number of vertices and  $m$  number of edges. So, this matrix  $Y$  of graph  $X$  is of  $n \times m$  matrix. According to the incidence matrix case, this matrix will have  $Y_{i,j} = 1$  if the vertex  $v_i$  and edge  $e_j$  are related also called incident and 0 if they are not related.

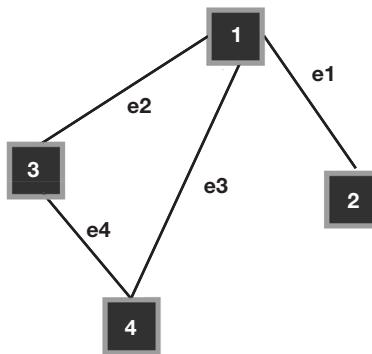


Figure 17.13 Undirected graph.

Table 17.3 shows the matrix based on the undirected graph shown in Figure 17.13.

Table 17.3 Incidence matrix for undirected graph

Sl. No.	e1	e2	e3	e4
1	1	1	1	0
2	1	0	0	0
3	0	1	0	1
4	0	0	1	1

- Directed graph:** As the name suggests, directed graph is one which contains vertices connected by edges. These edges carry a direction with them. The incidence matrix of a directed graph,  $D$ , is shown in Figure 17.14. It contains  $n$  number of vertices and  $m$  number of edges of matrix  $M$ . So, matrix  $M$  is an  $n \times m$  matrix, as shown in Table 17.4. This matrix content is set based on the directed relationship. For example, if edge  $e_j$  leaves vertex  $v_i$  then position  $M_{i,j} = -1$ , if it enters vertex  $v_i$  then 1 and 0 if it does not enter.

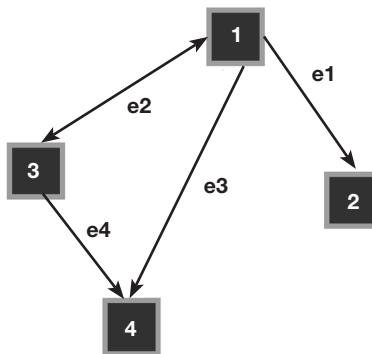


Figure 17.14 Directed graph.

Table 17.4 shows the matrix based on the directed graph shown in Figure 17.14.

**Table 17.4** Incidence matrix for directed graph

Sl. No.	e1	e2	e3	e4
1	-1	1	-1	0
2	1	0	0	0
3	0	1	0	-1
4	0	0	1	1

## Summary

Data structure is an important concept of programming. In any programming language, you will encounter at least one data structure that you will need to complete your program. There are various types of data structures. Some are very common, while others are rarely used but very useful in solving complex problems.

In this chapter, we have learned the following:

1. What is a data structure?
2. How to use data structure in programs.
3. Linear and non-linear data structures.
4. Benefits of using Binary Tree.
5. Difference between tree and graph.
6. Directed and weighted graph.
7. Adjacency matrix and adjacency list.

In Chapter 18, we will learn about lambdas and functional programming. We will spend time understanding lambdas and explore the ways to use it in a program. We will explore the use of functional programming.

## Multiple-Choice Questions

1. Which of the following has elements arranged to each other?
  - (a) Linear Data Structure
  - (b) Non-Linear Data Structure
  - (c) Both (a) and (b)
  - (d) None of the above
2. Which one of the following data types is utilized only for positive values in data structure?
  - (a) Arrays
  - (b) Unsigned
  - (c) Signed
  - (d) Boolean
3. What is the minimum number of nodes that a binary tree can possess?
  - (a) Two
4. You can access an array by referring to the indexed element within the array.
  - (a) True
  - (b) False
5. Which of the following trees has each internal node containing an operator and each leaf node containing an operand?
  - (a) Winner Tree
  - (b) Loser Tree
  - (c) Expression Tree
  - (d) Binary Search Tree

## Review Questions

1. What is data structure?
2. How do we use data structure?
3. What is linear data structure?
4. What is non-linear data structure?
5. What is the difference between linear and non-linear data structure?

6. Give at least two scenarios on when to use linear and when to use non-linear data structure.
7. Define tree data structure.
8. What are the frequently used subsets of tree data structure?
9. What is the search complexity of Binary Search Tree?
10. What is Expression Tree?
11. What is graph structure?
12. Give one real-life example of graph structure in use.
13. What is directed graph?
14. What is weighted graph?
15. How do you create adjacency matrix?
16. How do you create adjacency list?
17. Which graph representation is faster in finding the edge between two nodes?
18. What is the complexity in removing edge in adjacency matrix?

## Exercises

---

1. Plot your Facebook account data and figure out your personal graph like your friends, their friends, common friends, etc. Design a graph and add weight to it.
2. Create an adjacency matrix and adjacency list for the graph in Question 1.
3. Create a chart to show the differences, advantages, and disadvantages between linear and non-linear data structure.

## Project Idea

---

Design your own social media platform. Create pages to add users and let them search the platform to look for connections. Write a detailed plan on how to store and retrieve data. Figure out the best possible data structure for

this problem and design one. Collect as much as data and run various algorithms on it to learn how you can extend functionalities for your social network.

## Recommended Readings

---

1. Allen B. Downey. 2017. *Think Data Structures: Algorithms and Information Retrieval in Java*. O'Reilly Media: Massachusetts
2. James Cutajar. 2018. *Beginning Java Data Structures and Algorithms: Sharpen your problem solving skills by learning core computer science concepts in a pain-free manner*. Packt: Birmingham
3. Mr Kotiyana. 2018. *Introduction to Data Structures and Algorithms in Java*. [Independently Published]
4. Suman Saha and Shailendra Shukla. 2019. *Advanced Data Structures: Theory and Applications*. Chapman and Hall/CRC: London

# Lambdas and Functional Programming

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Functional programming.
- Higher-order and first-class functions.
- Pure functions and their characteristics.
- Recursion functions, evaluation, and referential transparency.
- Working examples of all functions using Java.
- Lambdas.

### 18.1 | Introduction

Programming applications in different languages require a specific style known as programming paradigm. In the earlier days, programmers used to code in C via procedural programming. Procedural programming is also called *imperative programming*. In a procedural paradigm, the code runs step-by-step, similar to kitchen recipes. The paradigm was excellent for its time, but as the complexities in the developing world grew, it was realized that there was a need for a reusable code that could make programming easier for all the stages of software lifecycle.

Companies revolutionized their business processes by adding a desktop or web system to achieve greater business productivity. Programmers soon realized that procedural programming was ineffective and cumbersome for their requirements.

As a result, object-oriented paradigm was introduced. In object-oriented programming (OOP), “objects” were introduced that mirrored the real world where each object can have its distinct behavior and state. Thus, developers began viewing code with respect to the real world and added functionalities accordingly.

OOP was globally instrumental in shrinking the size of codebases. As users were able to assign behavior and states to each object, they used OOP components such as inheritance, polymorphism, encapsulation, and abstract to achieve unforeseen levels of productivity. The programs that once required 5,000 lines of code were now reduced to only 500 lines.

OOP manages to resolve several issues, but with advancements in academic as well as industry landscapes, it was simply not enough to work in all types of environments. Thus, a newer programming paradigm achieved success and became popular for programming. This new paradigm, called *functional programming*, is supported by mainstream languages such as Python, PHP, and Java, while several languages such as Haskell are gaining prominence due to their effectiveness for functional-based applications.

### 18.2 | Functional Programming

As the name suggests, functional programming relies on “functions”. Here, functions can refer to any operation or feature that has to be added into the application. With functional programming, there is no requirement for modifying the other components of the code in the process of adding a new function.

These functions can be seen as similar to real-world formulas in mathematics. In fact, functional programming is heavily based on mathematical functions. In mathematics, a problem is tackled by reasoning and solving with a technique that the real theory behind all the complex steps of the function are abstracted, and the problem solver can focus more on higher-level complexities by designing a solution. Likewise, functional programming can be seen as providing a greater level of abstraction in software development.

Functional programming prevents modification in states and mutability of data. Instead of statements, expressions are used in functional programming.





How is functional programming different from object-oriented programming?

### 18.2.1 Fundamental Concepts of Functional Programming

Before writing functional paradigm-based code, it is necessary to familiarize ourselves with the following concepts for the production of quality code. Without properly interpreting the underlying theory behind functional programming, our implementation can suffer in the production of powerful code.

#### 18.2.1.1 Higher-Order and First-Class Functions

A higher-order function exists in both mathematical and computer science worlds. Generally, a higher-order function can be described as possessing any of the two properties.

1. The result type of the function is also a function. For instance:

```
// Here A is a function that takes an integer value of total
function A ( int total)
{
    // The function performs some processing and adds it to return another function
    return marks();
}
```

2. The function's arguments include a single function or multiple functions. For example, we have defined a function `areaofRectangle`. The function takes the argument in the form of another function `values` that saves length and breadth of the rectangle and passes it to the former function.

```
// Here areaofRectangle is a function that has taken another function values for the
calculation of area
public areaofRectangle (values)
{
    // Since values is returned so the function can be considered as a higher-order
    function.
    return values;
}
```

Other than higher-order functions, we also have to acquaint ourselves with first-class functions. First-class functions are similar to higher-order functions. However, in first-class function, it is mandatory to follow both the conditions of a higher-order function; that is, a first-class function must return another function and contain a function for its parameter arguments also. Thus, each first-class function can be termed as higher-order function.

However, there is a thin line that separates both types of functions. When we use the term “higher order”, our focus is more inclined towards the mathematical aspect of the problem, while “first class” is used to view the problem in a computer science oriented approach.

#### 18.2.1.2 Pure Functions

As already discussed, one of the core components of a functional application is the presence of expressions. These expressions are also called *pure functions*. They are beneficial because they do not incur any significant side effects on input/output operations or memory. In functional programming, a function has a side effect when it is able to change states of data that lie outside its scope. A pure function has the following characteristics:

1. Sometimes, an expression is useful for a while. However, with the passage of time newer changes to the application may render the expression as useless. In other paradigms where statements are used in place of expressions, removing these statements may have an adverse impact on the application. However, in functional programming by eliminating expressions from an application, there is no effect on the operation of any other expression.

2. In computer science, sometimes a certain strategy called *memoisation* is used for increasing the speed of computer programs. In memoisation, the results of a function are stored in the cache so they can be returned when the function is called using the same inputs. Pure functions support memoisation; that is, if a pure function uses the same inputs as arguments and is continuously called by the applications, the system resources can be saved by issuing the same result through caching.
3. When side effects are not supported by the complete language, then a compiler can process expressions by any evaluation technique.
4. Pure functions can employ parallelism. What this means is that if the data that is stored by two pure expressions is separate from any relationship (logical or data dependency), then both expressions can be simultaneously processed. Likewise, they can also be rearranged.

**QUICK CHALLENGE**

Give an example of pure function.

### 18.2.1.3 Recursion

In procedural and objective paradigms, looping (iteration) of elements is performed with the use of loops (for loop, while loop). On the other hand, functional languages are geared towards the use of recursion for looping.

Recursion is a programming concept in which a function calls (invokes) itself. A recursive function contains conditions that repeat calling the function until it reaches to a base case. Some people believe that using loops or recursion is a matter of preference. However, recursion is productive in many cases where it:

1. Reduces the lines of code.
2. Reduces the possibility of errors.
3. Reduces the cost of the function.

Recursion techniques can be used by higher-order functions by utilizing anamorphisms and catamorphisms. As a result, the higher-order functions contribute in the development of control structures (like loops) in procedural programming languages.

Usually, functional languages support unrestricted recursions and are also equipped with Turing complete. Thus, the halting problem is undecidable; that is, an issue cannot be responded with a yes or no conclusion. A halting problem is one in which there is an issue of determining whether a computer program (containing an input) has to be stopped or allowed to run for an infinite period of time.

Recursion in functional programming paradigms also need some sort of “inconsistency” that is added into the logic of the language’s *type system*. (The set of rules that adds the type property to a construct in a programming language is called type system.) A few functional languages limit recursion types and may allow only the use of well-founded recursion.

**QUICK CHALLENGE**

Define Anamorphisms and Catamorphisms.

### 18.2.1.4 Evaluation

A function language can be grouped by two factors – strict and non-strict. These factors are influenced by the language’s ability to evaluate an expression with the arguments of the function. To understand strict evaluation and non-strict evaluation, let us take the following example where we have to print length of 5 elements:

```
print length([2-1, 5+7, 1/0, 8*8, 4+9])
```

If this expression gets evaluated by a non-strict evaluation, then a length of 5 is returned (as there are 5 elements in the list). Non-strict evaluation does not attempt to go into the intricacies of the elements. On the other hand, if strict evaluation is used, then the expression does not succeed because it finds an error in the third element of the list (1/0). Thus, strict evaluation delves into the details of expression to determine the logic and semantics of the elements. However, when the elements of a function are required for evaluation due to some calling, then non-strict evaluation assesses them in the same manner as strict evaluation.

Non-strict evaluation is implemented using graph reduction. Languages such as Haskell and Clean adopt non-strict evaluation.

### 18.2.1.5 Referential Transparency

Assignments statements are not supported in a functional program. What this means is that when a variable is assigned and defined with a value, then it cannot be modified in the future. Due to this property, the side effects of an application are reduced to a significant extent as the variables' values are changeable in the duration of execution processes. Hence, functional programs can be considered as referentially transparent.

Suppose there is a basic program in C that has the following expression, where the value of  $a$  keeps getting changed because of successive evaluations:

```
a = a * 20
```

Let us consider the initial value of  $a$  as 1. Now, after a single evaluation the value of  $a$  becomes 20. An additional evaluation can increase the value of  $a$  to 400. Since using any of these values changes the program's semantics, the expression cannot be said as referentially transparent. This is because its actual value is being continuously changed implicitly. In functional programming, the above example can be changed into the following function:

```
int addone(int a) {return a+2;}
```

Now, this evaluation always returns a fixed answer as the value of  $a$  is not modified implicitly. Likewise, the function does not incur with any side effect. Hence, it can be said that functional programs are inherently referentially transparent.

#### QUICK CHALLENGE

Give an example of referential transparency.

## 18.3 | Functional Programming in Java

Due to the growing demand of the functional paradigm in development circles, the release of Java 8 and Java 9 came with extensive support for functional programming. While functional programming was still possible in earlier versions of Java, there were too many restrictions to take Java seriously. However, with Java 8 and the recent Java 9 release, Oracle has transformed Java as a formidable option for functional enthusiasts. Let us visit some of the functional aspects in Java.



What are the other languages that support functional programming? What advantage does Java have over them?

### 18.3.1 Pure Function

Earlier we talked about pure function as a general concept. In Java 9, a pure function can be written as:

```
public class PureFnObject{
// the values of x and y are taken as input arguments
    public int subtract(int x, int y) {
        // The function returns the subtracted answer
        return x - y;
    }
}
```

There are two factors that make this function a pure one. First, you can observe that the subtract function is entirely reliant on the input arguments. Second, there are no side effects of the function because the values of  $x$  and  $y$  can be only changed inside the function. On the other hand, the same example can be written in the following code where the function is non-pure.

```
public class NonpureFnObject{
    private int x = 0;
    public int subtract(int x) {
        this.x -= x;
        return this.x;
    }
}
```

In this example, the value of the variable *x* is calculated by using the member variable, which means it can be easily modified. Thus, it is changeable and the function can be said to be carrying a side effect.

**QUICK CHALLENGE**

Create a chart to show the difference between pure and non-pure functions.

### 18.3.2 Higher-Order Function

In Java, a higher-order function can be replicated by adding a single or multiple lambda expressions in the parameters of a function. Additionally, the function must return a lambda expression. Let us see the following example.

```
public class HigherOrderFn {
    public <T> IFactory<T> createFactory(IProducer<T> prod, IConfigurator<T> conf) {
        // the lambda expression begins
        return () -> { // the arrow represents the lambda expression
            T ins = prod.produce();
            conf.configure(ins);
            return ins;
        } // the lambda expression ends
    }
}
```

Observe that a lambda expression is returned by method of `createFactory()`. For this reason, the *t* condition of a higher-order function is verified.

### 18.3.3 No State

There are no states in a functional program. Here, state means that the member cannot point (reference) to a variable of the object or class. Thus, it can only reference its own variables. For instance, a function with an external state is given as follows:

```
public class Subtract {
    private int beginningValue = 2;
    public int sub(int x) {
        // the original value is modified by addition
        return beginningValue + x;
    }
}
```

Now, observe another example where a function does not have an external state. Here, the values of the function cannot be changed.

```
public class Subtract {
    public int sub(int x, int y) {
        return x + y;
    }
}
```

### 18.3.4 Functional Interfaces

In Java, an interface with a single abstract method is called functional interface. As you know, abstract methods are those that are incomplete and are not implemented. Interfaces can have both static and default methods (implemented). A functional interface must have a single method which is not implemented. For example:

```
public interface TestingInterface {
    public void run();
}
```

Let's observe the following program:

```
public interface TestingInterface2 {
    public void run();
    public default void hello() {
        System.out.println("hello world");
    }
    public static void helloStatic() {
        System.out.println("static hello world");
    }
}
```

You must be thinking that since two methods that have been implemented, the interface may not be functional. But since the run method is abstract (i.e., it has not been implemented), our interface is still functional. Interfaces that have multiple abstract methods cannot be categorized as a functional interface.

## 18.4 | Object-Oriented versus Functional Programming



So far you have learnt object-oriented and functional programming. Now let us do the head-to-head comparison between them. Table 18.1 compares these two in multiple areas.

**Table 18.1** Object-oriented and functional programming comparison

Parameter	Functional Programming	Object-Oriented Programming
<b>Focus</b>	It focuses on evaluation of functions	It focuses on objects
<b>Data</b>	It mainly uses immutable data	It mainly uses mutable data
<b>Model</b>	It follows declarative programming model	It follows imperative programming model
<b>Parallel Programming</b>	It supports parallel programming	It does not support parallel programming
<b>Execution</b>	Statements executed in any order	Statements executed in a specific order
<b>Iteration</b>	It uses recursion	It uses loops
<b>Elements</b>	Variables and functions	Objects and methods
<b>Use</b>	Used when there are a few things with large number of operations	Used when there are large number of things with a few operations
<b>State</b>	It does not care about any state	It does care about the state

(Continued)

**Table 18.1** (Continued)

Parameter	Functional Programming	Object-Oriented Programming
<b>Primary Unit</b>	Function	Object
<b>Abstraction Support</b>	It supports abstraction over data and behavior	It supports abstraction over data only
<b>Performance on Big Data Processing</b>	Very good	Not so good
<b>Conditional Statements</b>	No support	Supports statements like if-else, switch, etc.
<b>Example</b>	<pre>employees.stream().filter(emp -&gt; emp.salary &lt; minSalary).forEach(emp -&gt; System.out. println(emp));</pre>	<pre>for (Employee emp: employees) {     if (emp.getSalary() &lt; minSalary) } System.out.println(emp); }</pre>



How is Stream different from Collection?

## 18.5 | Lambdas

Now that you have understood the concept of functional interfaces, let us proceed to lambdas. When Java 8 was released, it gained significant traction with the introduction of lambdas. So what exactly are lambdas expressions?

Lambda expressions are an attempt by Java to enhance functional programming. A lambda expression assists a single or multiple instances of a functional interface via concrete implementations. Since lambdas are capable for concrete implementation of a single function, they are appropriate for functional interfaces. They can be created without necessarily adding them to a class. Lambda expressions can be treated as an object and can be passed and executed like an object. The format of a lambda expression is as follows:

parameter -> the body of expression

Here, the arrow operation is used to represent it.



Is using lambda faster than traditional Java programming?

### 18.5.1 Elements of Lambda Expression

Following are the main elements you need to know in order to code in Lambda.

#### 18.5.1.1 Parameters

In a typical lambda expression which consists of multiple parts and each part is connected via an arrow. We will learn more about the meaning of these parts in the subsequent sections. On the left side, we can have  $n$  number of parameters, where  $n$  can also be a 0. Adding the type of parameter is optional. The type of parameter can be determined by the compiler by sifting through the coding context. Multiple parameters are entailed in round brackets, (). For single parameters, the use of a round brackets is optional. If there is no parameter, then it can be represented by an empty round bracket.

#### 18.5.1.2 Body

Similar to the parameters, the body may also carry  $n$  number of statements. These statements are represented by curly brackets, {}. However, a single statement does not require curly brackets. The body expression also embodies the return type of the function.

Before going into lambdas, let us check the following example:

```
package java11.fundamentals.chapter18;
public class SimpleMethodExample {

    public void printDemo(String simpleText) {
        // A method to print the String
        System.out.println(simpleText);
    }
    public static void main(String[] args) {
        // Creating an instance of the object for our LamdaExample class
        SimpleMethodExample smEx = new SimpleMethodExample();

        // Assign a value to the object's string
        String simpleText = "A Simple String";
        smEx.printDemo(simpleText);
    }
}
```

The above program produces the following result.

### A Simple String

You may have found the above example as a traditional approach where the implementation of the method is kept hidden from the caller. Here, the caller takes a variable which is then passed to a method `printDemo`. As you can see, there is a side effect issue.

Similarly, let us visit another example in which we are passing a behavior instead of a variable. We have utilized a functional interface for this example.

```
package java11.fundamentals.chapter18;
public class LambdaFunctionalInterfaceExample {

    interface printingSomeText {
        void print(String anyValue);
    }
    public void printLambdaText(String lambdaText, printingSomeText pst) {
        pst.print(lambdaText);
    }
    public static void main(String[] args) {
        LambdaExample lmd1 = new LambdaExample();
        String lambdaText = "Understanding Lambdas";
        printingSomeText pst = (String letsPrint)->{System.out.println(letsPrint);};
        lmd1.printLambdaText(lambdaText, pst);
    }
}
```

The above code produces the following result.

### Understanding Lambdas

As you can observe, all the complexity behind the execution of printing the string has been handled by the interface. The method `printLambdaText` was used in the example only for the execution of the interface's block of code. Let us code the above program further:

```

package java11.fundamentals.chapter18;
public class LambdaFunctionalInterfaceExample2 {
    interface printingSomeText {
        void print(String anyValue);
    }

    public void printLambdaText(String lambdaText, printingSomeText pst) {
        pst.print(lambdaText);
    }
    public static void main(String[] args) {
        LambdaFunctionalInterfaceExample2 lmd1 = new LambdaFunctionalInterfaceExample2();
        String lambdaText = "Understanding Lambdas";

        printingSomeText pst = new printingSomeText() {
            @Override
            public void print(String anyValue) {
                System.out.println(anyValue);
            }
        };

        lmd1.printLambdaText(lambdaText, pst);
    }
}

```

The above program produces the following result.

### Understanding Lambdas

Here, we wrote an implementation for our interface, which was then passed to the `printLambdaText` method. This example was necessary because it can help us realize why we need lambdas. Now, by introducing lambdas, we can recode by writing the following:

```

package java11.fundamentals.chapter18;
public class LambdaFunctionalInterfaceExample3 {

    interface printingSomeText {
        void print(String anyValue);
    }

    public void printLambdaText(String lambdaText, printingSomeText pst) {
        pst.print(lambdaText);
    }

    public static void main(String[] args) {
        LambdaFunctionalInterfaceExample3 lmd1 = new
        LambdaFunctionalInterfaceExample3();
        String lambdaText = "Understanding Lambdas";
        printingSomeText pst = (String letsPrint) -> {
            System.out.println(letsPrint);
        };
        lmd1.printLambdaText(lambdaText, pst);
    }
}

```

The above program produces the following result.

### Understanding Lambdas

Does this not look better? By adding just a single line of a lambda expression, we have improved our code. Lambdas took the parameter and handled its mapping according to our method's parameter list. The code after the arrow can be seen as a concrete implementation of this method.

### 18.5.2 Examples

Let us code more examples to gain a better understanding of lambda expressions. We will implement the next example with the help of lambda expression, where we have an interface that draws a square.

```
package java11.fundamentals.chapter18;
interface drawSquare {
    public void drawIt();
}
public class LambdaDrawSquareExample {
    public static void main(String[] args) {

        int area = 5;

        // Lambda expression starts. An instance ds of the interface is taken as a parameter.
        Note that the absence of the parameters is represented by an empty round bracket
        drawSquare ds = () -> {
            System.out.println("The square is drawn for the area " + area);
        };

        // The concrete implementation provided by the lambda expression draws the square
        from the functional interface's method
        ds.drawIt();

    }
}
```

The above program produces the following result.

**The square is drawn for the area 5**

#### 18.5.2.1 Lambda Expression without Parameters

We have already mentioned that a lambda parameter can be 0. We demonstrate this with the help of the following example:

```
package java11.fundamentals.chapter18;
public class LambdaExpressionWithoutParametersExample {
    interface announcement {
        public String announce();
    }

    public static void main(String[] args) {

        // Lambda expression begins
        announcement a = () -> {
            return "The flight going to New York has been cancelled due to the extreme weather";
        };
        // Lambda Expression ends

        System.out.println(a.announce());
    }
}
```

The above program produces the following result.

**The flight going to New York has been cancelled due to the extreme weather**

### 18.5.2.2 Adding a Single Parameter to the Lambda Expression

Now, let us see how we can use a parameter by continuing with our announcement example.

```
package java11.fundamentals.chapter18;

public class LambdaExpressionWithSingleParameterExample {

    interface announcement {
        // Adding a parameter to the method with a String variable
        public String announce(String annText);
    }

    public static void main(String[] args) {
        // First lambda expression begins
        announcement a1 = (annText) -> { // adding a single parameter with optional round
brackets
            return "We have an important announcement to be made. " + annText;
        }; // First lambda expression ends
        System.out.println(a1.announce("The flight going to New York has been cancelled
due to the extreme weather "));
        // Second lambda expression begins
        announcement a2 = annText -> { // adding a single parameter without using round
brackets
            return "We have an important announcement to be made. " + annText;
        };
        // Second lambda expression ends
        System.out.println(a2.announce("The flight going to Boston has been cancelled due
to a hailstorm "));
    }
}
```

The above program produces the following result.

We have an important announcement to be made. The flight going to New York has been cancelled due to the extreme weather  
We have an important announcement to be made. The flight going to Boston has been cancelled due to a hailstorm

### 18.5.2.3 Lambda Expression with Multiple Parameters

Let us consider that we have to add the total marks of 5 subjects for 5 students. Now, we have an interface for the addition of marks where the method contains the value for subjects. Here, lambda expression can prove beneficial by providing convenience for concrete implementations. Since we have more than a single parameter, we must use round brackets in our expressions. Unlike single parameters, we cannot optionally remove brackets.

```

package java11.fundamentals.chapter18;

public class LambdaExpressionWithMultipleParametersExample {

    interface reportCard {
        // Adding multiple parameters to the method
        public int marksForSubjects(int mathematics,int physics,int biology,int history,
int chemistry);
    }

    public static void main(String[] args) {
        reportCard am1=(int mathematics,int physics,int biology,int history, int
chemistry)->(mathematics + physics + biology + history+ chemistry); // round brackets
are used for multiple parameters
        System.out.println(am1.marksForSubjects(74,87,66,53,90));

        reportCard am2=(int mathematics,int physics,int biology,int history, int
chemistry)->(mathematics + physics + biology + history+ chemistry);
        System.out.println(am2.marksForSubjects(40,40,50,60,70));

        reportCard am3=(int mathematics,int physics,int biology,int history, int
chemistry)->(mathematics + physics + biology + history+ chemistry);
        System.out.println(am3.marksForSubjects(50,60,70,60,70));

        reportCard am4=(int mathematics,int physics,int biology,int history, int
chemistry)->(mathematics + physics + biology + history+ chemistry);
        System.out.println(am4.marksForSubjects(64,68,71,67,70));

        reportCard am5=(int mathematics,int physics,int biology,int history, int
chemistry)->(mathematics + physics + biology + history+ chemistry);
        System.out.println(am5.marksForSubjects(85,86,55,75,88));
    }
}

```

The above program produces the following result.

370
260
310
340
389

### 18.5.3 Using return

Now that we have learned how to use various parameters for our use cases, let us move on to the `return` keyword. When a lambda expression contains a single statement, then it is optional to explicitly add or ignore the addition of `return` keyword. However, if there are multiple expressions in the lambda expression, then it is important to add the `return` keyword or you may face an error. Now, going by the above example of adding the marks of students, we can rewrite the program by adding the `return` keyword.

```

package java11.fundamentals.chapter18;
public class LambdaExpressionUsingReturnExample {
    interface reportCard {
        // Adding multiple parameters to the method
        public int marksForSubjects(int mathematics,int physics,int biology,int history,
int chemistry);
    }
    public static void main(String[] args) {
        reportCard am1=(int mathematics,int physics,int biology,int history, int
chemistry)->{
            return (mathematics + physics + biology + history+ chemistry); // adding the
optional return keyword
        };
        System.out.println("The total of the first student is "+am1.
marksForSubjects(74,87,66,53,90));
        reportCard am2=(int mathematics,int physics,int biology,int history, int
chemistry)->{
            return (mathematics + physics + biology + history+ chemistry);
        };
        System.out.println("The total of the second student is "+am2.
marksForSubjects(40,40,50,60,70));
        reportCard am3=(int mathematics,int physics,int biology,int history, int
chemistry)->{
            return (mathematics + physics + biology + history+ chemistry);
        };
        System.out.println("The total of the third student is "+am3.
marksForSubjects(50,60,70,60,70));
        reportCard am4=(int mathematics,int physics,int biology,int history, int chemistry)->{
            return (mathematics + physics + biology + history+ chemistry);
        };
        System.out.println("The total of the fourth student is "+am4.
marksForSubjects(65,56,95,65,78));
        reportCard am5=(int mathematics,int physics,int biology,int history, int
chemistry)->{
            return (mathematics + physics + biology + history+ chemistry);
        };
        System.out.println("The total of the fifth student is "+am5.
marksForSubjects(85,86,55,75,88));
    }
}

```

The above program produces the following result.

**The total of the first student is 370  
The total of the second student is 260  
The total of the third student is 310  
The total of the fourth student is 359  
The total of the fifth student is 389**

Now, let us modify this example by adding multiple statements in each lambda expression. We are adding a statement that can increase the marks of the subject of mathematics by 10 for each student. Without the `return` keyword, these expressions cannot be run.

```

package java11.fundamentals.chapter18;
public class LambdaExpressionWithMultipleParametersExample {
    interface reportCard {
        // Adding multiple parameters to the method
        public int marksForSubjects(int mathematics, int physics, int biology, int
history, int chemistry);
    }
    public static void main(String[] args) {
        // Multiple parameters in lambda expression
        reportCard am1 = (int mathematics, int physics, int biology, int history, int
chemistry) -> {
            mathematics += 10;
            return (mathematics + physics + biology + history + chemistry); // mandatory
inclusion of return
        };
        System.out.println("The total of the first student is " + am1.marksForSubjects(74,
87, 66, 53, 90));
        reportCard am2 = (int mathematics, int physics, int biology, int history, int
chemistry) -> {
            mathematics += 10;
            return (mathematics + physics + biology + history + chemistry);
        };
        System.out.println("The total of the second student is " +
am2.marksForSubjects(40, 40, 50, 60, 70));
        reportCard am3 = (int mathematics, int physics, int biology, int history, int
chemistry) -> {
            mathematics += 10;
            return (mathematics + physics + biology + history + chemistry);
        };
        System.out.println("The total of the third student is " + am3.marksForSubjects(50,
60, 70, 60, 70));
        reportCard am4 = (int mathematics, int physics, int biology, int history, int
chemistry) -> {
            mathematics += 10;
            return (mathematics + physics + biology + history + chemistry);
        };
        System.out.println("The total of the fourth student is " +
am4.marksForSubjects(65, 56, 95, 65, 78));
        reportCard am5 = (int mathematics, int physics, int biology, int history, int
chemistry) -> {
            mathematics += 10;
            return (mathematics + physics + biology + history + chemistry);
        };
        System.out.println("The total of the fifth student is " + am5.marksForSubjects(85,
86, 55, 75, 88));
    }
}

```

The above program produces the following result.

The total of the first student is 380
The total of the second student is 270
The total of the third student is 320
The total of the fourth student is 369
The total of the fifth student is 399

#### 18.5.4 Lambdas with Loops

Lambdas are flexible and can also be integrated with loops. Let us consider two students, where each student is assigned the marks of 5 subjects. By using lambdas, we can use the **foreach** loop to add a print statement with each addition.

```
package java11.fundamentals.chapter18;
import java.util.*;
public class LambdaLoopExample {
    public static void main(String[] args) {
        List<Integer> student1 = new ArrayList<Integer>();
        student1.add(50);
        student1.add(60);
        student1.add(70);
        student1.add(80);
        student1.add(90);
        System.out.println("The marks of each subject of student1 :");
        student1.forEach((x) -> System.out.println(x));

        List<Integer> student2 = new ArrayList<Integer>();
        student2.add(90);
        student2.add(80);
        student2.add(70);
        student2.add(60);
        student2.add(50);
        System.out.println("The marks of each subject of student2 :");
        student2.forEach((x) -> System.out.println(x));
    }
}
```

The above program produces the following result.

```
The marks of each subject of student1 :
50
60
70
80
90
The marks of each subject of student2 :
90
80
70
60
50
```

#### 18.5.5 Lambdas with Threads

We can also associate lambdas expressions with threads. Consider you have four threads that have different behaviors. Now, we will code the first thread without lambda expression. Observe how the second thread does the same thing with a better strategy using lambdas.

```

package java11.fundamentals.chapter18;
public class LambdaThreadExample {
    public static void main(String[] args) {
        Runnable run1 = new Runnable() {
            public void run() {
                System.out.println("The first thread is currently in the state of running");
            }
        };
        Thread t1 = new Thread(run1);
        t1.start();

        Runnable run2 = () -> {
            System.out.println("The second thread is currently in the state of running");
        };
        Thread t2 = new Thread(run2);
        t2.start();
        Runnable run3 = () -> {
            System.out.print("The id of the third thread is ");
            System.out.println(Thread.currentThread().getId());
        };
        Thread t3 = new Thread(run3);
        t3.start();

        Runnable run4 = () -> {
            System.out.print("The class of the fourth thread is ");
            System.out.println(Thread.currentThread().getClass());
        };
        Thread t4 = new Thread(run4);
        t4.start();
    }
}

```

The above program produces the following result.

**The first thread is currently in the state of running  
 The second thread is currently in the state of running  
 The id of the third thread is 14  
 The class of the fourth thread is class java.lang.Thread**

## 18.6 | Date and Time API

The recent Java releases have improved the Date and Time API for tackling earlier issues that mainly existed in `java.util.Calendar` and `java.util.Date`.

### 18.6.1 Problems with Earlier APIs

In the past, developers complaint about the following conundrums:

1. Older versions of `Calendar` and `Date` APIs were found to be non-safe for threads. What this meant is that Java programmers had to continuously keep adding lines of code. Similarly, there were also issues pertaining to debug concurrency. Thus, it was problematic to debug several segments of the API simultaneously. The newer APIs in Java 9 for Date and Time are a game changer because they have been modified to be safe for threads while they are also plugged with immutability.
2. Older versions of `Calendar` and `Date` APIs lacked in design. Their built-in methods were ineffective in running standard tasks for applications. The Date and Time API of Java 9 is said to be ISO centric. All the values pertaining to time,

duration, periods, and date are handled through the use of consistent models. Likewise, standard tasks for daily applications can be performed easily via newer methods.

3. Earlier, time and resources went into the coding of logic related to the time zones of different areas. Newer APIs have automated this task through Local and ZonedDateTime APIs.

## 18.6.2 Local Date

The API of LocalDate adheres to the following ISO format:

```
yyyy-MM-dd
```

In the real world, such APIs are used in payroll systems for assigning a payroll to an employee at the beginning or end of the month. Likewise, in applications for storing birthdays such as in social media platforms, there are also requirements for dates. In case you require the latest date, type the following:

```
import java.time.LocalDate;
public class DateExample{
    public static void main(String args[]) {
        LocalDate currentDate = LocalDate.now();
        System.out.println(currentDate);
    }
}
```

The following output will be displayed.

2018-10-12

In order to get the result for any year, month, and day, you can use `of()` method. Similarly, `parse()` method can also be used. Let us see what happens if we use `of()` method:

```
import java.time.LocalDate;
public class DateExample{
    public static void main(String args[]) {
        System.out.println(LocalDate.of(2018, 01, 01));
    }
}
```

By using `parse()` method we can get the same answer:

```
import java.time.LocalDate;
public class DateExample{
    public static void main(String args[]) {
        System.out.println(LocalDate.parse("2018-01-01"));
    }
}
```

In both instances, we get the following answer.

2018-01-01

Likewise, let us go through a few more methods related to dates. Suppose you are developing an application which assigns tasks to employees. Now, an employee can be given a task that has to be done by the following day. In order to add this functionality in the code, you can write the following:

```

import java.time.LocalDate;
public class DateExample{
    public static void main(String args[]) {
        LocalDate taskDate = LocalDate.now().plusDays(1);
        System.out.println(taskDate);
    }
}

```

As such, you can change the values in the method to get your desired date in future.

Now, suppose you wish to go exactly one month back for your application requirements. This can be done through the `minus()` method.

```

import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
public class DateExample {
    public static void main(String args[]) {
        LocalDate lastMonth = LocalDate.now().minus(1, ChronoUnit.MONTHS);
        System.out.println(lastMonth);
    }
}

```

Consider the following example, where you have to check the date of a specific day. You can parse your date and utilize the method `.getDayOfWeek()` for displaying a date.

```

import java.time.DayOfWeek;
import java.time.LocalDate;
public class DateExample {
    public static void main(String args[]) {
        DayOfWeek whichDay = LocalDate.parse("2018-03-10").getDayOfWeek();
        System.out.println(whichDay);
    }
}

```

Similarly, the day of a month can be returned by using `.getDayOfMonth()` method, as follows:

```

import java.time.DayOfWeek;
import java.time.LocalDate;
public class DateExample {
    public static void main(String args[]) {
        int dayOftheMonth = LocalDate.parse("2018-03-10").getDayOfMonth();
        System.out.println(dayOftheMonth);
    }
}

```

In time related APIs, sometimes circumstances demand the need of identifying a leap year. Leap years can be checked by writing the following code:

```
import java.time.DayOfWeek;
import java.time.LocalDate;
public class DateExample {
    public static void main(String args[]) {
        boolean isItaLeapYear = LocalDate.now().isLeapYear();
        System.out.println(isItaLeapYear);
    }
}
```

For comparing two dates to check when a date occurred in comparison to another date, we can write:

```
import java.time.DayOfWeek;
import java.time.LocalDate;
public class DateExample {
    public static void main(String args[]) {
        boolean beforeOrNot = LocalDate.parse("2018-06-13")
            .isBefore(LocalDate.parse("2018-06-10"));
        System.out.println(beforeOrNot);
        boolean afterOrNot = LocalDate.parse("2018-06-10")
            .isAfter(LocalDate.parse("2018-06-09"));
        System.out.println(afterOrNot);
    }
}
```

### 18.6.3 LocalTime

The `LocalTime` class returns time. It works similar to the `LocalTime` class. For getting the current time, write the following:

```
import java.time.LocalTime;
public class TimeExample{
    public static void main(String args[]) {
        LocalTime whatIsTheTime = LocalTime.now();
        System.out.println(whatIsTheTime);
    }
}
```

To parse time, you can write:

```
import java.time.LocalTime;
public class TimeExample{
    public static void main(String args[]) {
        LocalTime parsingTime = LocalTime.parse("03:20");
        System.out.println(parsingTime);
    }
}
```

The same thing can be done by using `of()` method:

```
import java.time.LocalTime;
public class TimeExample{
    public static void main(String args[]) {
        LocalTime usingOf = LocalTime.of(3,20);
        System.out.println(usingOf);
    }
}
```

To add time, you can use the `.plus()` method. We have used this method to add 5 hours to the current time.

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
public class TimeExample{
    public static void main(String args[]) {
        LocalTime fastForward = LocalTime.parse("03:20").plus(5, ChronoUnit.HOURS);
        System.out.println(fastForward);
    }
}
```

To get the hour, we can write:

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
public class TimeExample{
    public static void main(String args[]) {
        int whichHour = LocalTime.parse("03:20").getHour();
        System.out.println(whichHour);
    }
}
```

To compare the time, we can write:

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
public class TimeExample{
    public static void main(String args[]) {
        boolean comparingTime = LocalTime.parse("03:20").isBefore(LocalTime.
parse("02:30"));
        System.out.println(comparingTime);
    }
}
```

Sometimes in DB queries, records are required according to a given time period. To obtain such records, we can get time for noon, minimum, and maximum:

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
public class TimeExample{
    public static void main(String args[]) {
        LocalTime maximumTime = LocalTime.MAX;
        System.out.println(maximumTime);
    }
}
```

#### 18.6.4 LocalDateTime

While the above classes are useful for specific cases pertaining to date and time, sometimes both values are required (i.e., a date as well as the exact time for that day). For such scenarios, `LocalDateTime` is used. Now, let us go through its methods. To get the current date and time, we have to write:

```
import java.time.LocalDateTime;
import java.time.LocalTime;
public class TimeExample{
    public static void main(String args[]) {
        System.out.println(LocalDateTime.now());
    }
}
```

To use `of()` method, we write:

```
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
public class TimeExample{
    public static void main(String args[]) {
        System.out.println(LocalDateTime.of(2018, Month.MARCH, 10, 03, 30));
    }
}
```

The same thing can also be done for parsing:

```
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
public class TimeExample{
    public static void main(String args[]) {
        System.out.println(LocalDateTime.parse("2018-01-20T06:24:00"));
    }
}
```

For adding and subtracting time, we can use `plus()` and `minus()` methods, just as we have been using them previously.

```

import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
public class DateAndTimeExample {
    public static void main(String args[]) {
        LocalDateTime addHours = LocalDateTime.now().plusHours(3);
        System.out.println(addHours);
    }
}

```

Now, let us see the minus example.

```

import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
public class DateAndTimeExample {
    public static void main(String args[]) {
        LocalDateTime subHours = LocalDateTime.now().minusHours(3);
        System.out.println(subHours);
    }
}

```

Lastly, for getting a specific month, we can simply write:

```

import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.Month;
public class DateAndTimeExample {
    public static void main(String args[]) {
        System.out.println(LocalDateTime.now().getMonth());
    }
}

```

### 18.6.5 ZonedDateTime API

In order to combat the issue of time zone, we can use `ZonedDateTime()` in Java 9. In this API, an identifier called `ZoneID` represents time zones. To create the time zone of any specific city, type the following code in the IDE:

```

import java.time.ZoneId;
public class TimeZoneExample {
    public static void main(String args[]) {
        ZoneId id = ZoneId.of("Asia/Seoul");
        System.out.println(id);
    }
}

```

To get all the time zones that are listed in the API, we can simply write:

```
import java.time.ZoneId;
import java.util.Set;
public class TimeZoneExample {
    public static void main(String args[]) {
        Set<String> allIds = ZoneId.getAvailableZoneIds();
        System.out.println(allIds);
    }
}
```

To choose a specific time zone, we can write:

```
ZonedDateTime specificZone = ZonedDateTime.of(localDateTime, zoneId)
```

Now, let's create a localDateTime:

```
import java.time.LocalDateTime;
import java.time.Month;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.Set;
public class TimeZoneExample {
    public static void main(String args[]) {
        LocalDateTime ltd = LocalDateTime.of(2018, Month.MARCH, 10, 07, 20);
        System.out.println(ltd);
    }
}
```

Now, it is possible to add four hours and create a ZoneOffset in the above example. Let us continue the code.

```
import java.time.LocalDateTime;
import java.time.Month;
import java.time.OffsetDateTime;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.ZonedDateTime;
import java.util.Set;
public class TimeZoneExample {
    public static void main(String args[]) {

        LocalDateTime ltd = LocalDateTime.of(2018, Month.MARCH, 10, 07, 20);
        ZoneOffset os = ZoneOffset.of("+04:00");
        OffsetDateTime osbyfour = OffsetDateTime.of(ltd, os);
        System.out.println(osbyfour);
    }
}
```

The result of localDateTime of method is as follows

2018-03-10T07:20+04:00



Can DateTimeAPI get a user's local date?

## Summary

---

Functional programming is a practice of programming as functions like mathematical functions. Lambda expression is as a way of supporting functional programming in Java. This language is a declarative one, which means logic is expressed without its control flow.

In this chapter, we have learned the following concepts:

1. Functional programming and lambda.
2. Higher order and first order functions.
3. Pure functions and how to use them.
4. Recursion and how to use it in program.

In Chapter 19, we will learn about multithreading and reactive programming. We will learn about the multithreading world and understand the important concepts of concurrency. We will explore various examples and learn about deadlocks, synchronization blocks, lazy initialization, etc.

## Multiple-Choice Questions

---

1. We need to compile Lambda expression to anonymous inner classes.
  - (a) True
  - (b) False
2. The filter method in Stream API takes in a \_\_\_\_\_ as argument.
  - (a) Predicate
  - (b) Consumer
  - (c) Function
  - (d) Supplier
3. Functional interfaces can have more than one default methods.
- (a) (b) (c) (d)
4. Which of the following is used to get only the current time?
  - (a) LocalDate.now();
  - (b) LocalTime.now();
  - (c) LocalDate.now()
  - (d) All of the above
5. Can lambda expression accept multiple parameters?
  - (a) Yes
  - (b) No

## Review Questions

---

1. What is lambda?
2. How does functional programming work?
3. What are the benefits of using lambdas?
4. How do we print date in a local format?
5. How do we print date according to time zone?
6. How do we use lambdas with threads?
7. How do we use lambdas with loops?

## Exercises

---

1. Write a program to print stock prices of top 10 companies using lambda and functional programming.
2. Write a program to use DateTime API to print minute-by-minute status of a football match.
3. Create a chart to present all the benefits of using lambdas and functional programming.

## Project Idea

---

Create a chat application using lambda and functional programming. Capture each conversation and use DateTime API to print the conversation as per the user's local time. Consider user's time zone in this case.

## Recommended Readings

---

1. Kishori Sharan. 2018. *Java Language Features: With Modules, Streams, Threads, I/O, and Lambda Expressions*. Apress: New York
2. Pierre-Yves Saumont. 2017. *Functional Programming in Java: How Functional Techniques Improve Your Java Programs*. Dreamtech Press: New Delhi
3. Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft. 2018. *Modern Java in Action: Lambdas, streams, functional and reactive programming*. Manning Publications: New York



# Multithreading and Reactive Programming

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Reactive programming.
- Multithreading and how to program.
- Concurrency API improvements.
- How to deal with individual threads.
- Synchronization blocks.
- Deadlocks and how to resolve deadlocks.
- Concurrent data structures.
- How to design concurrent Java programs.
- Controlling sequential executions.
- How to avoid lazy initialization.

### 19.1 | Introduction

Java 11 is the latest version of Java released by Oracle, which has some interesting features. Before its release, Java 9 brought major updates. Java 9 updates offer improved support to large heaps as there is great demand for improving the capacity of Java to handle large memory sizes that are required to support cloud applications. It follows a module system and includes several Java projects that were initially defined for the new release.

Our focus in this chapter remains on discussing the improved multithreading capabilities of Java. We will begin this chapter by describing concepts such as reactive programming and multithreading, especially with their application using the new and improved Java's concurrency utilities. We will also cover thread transition, thread interaction, and `newworkstealingPool()` method. With the exercises and other tools provided in this chapter, we believe that Java programmers will learn to use multiple threads in their programming and ensure that their programs are efficient by employing the available capacities in modern processors.

### 19.2 | Reactive Programming



Reactive programming is a special method that employs an asynchronous style. It refers to a method that employs improved control over data streams and data streams use in changing the way the programming behaves with the future data stream. All kinds of data streams can be expressed using reactive methods and the programs can be designed to execute different changes automatically, according to the data flow that they receive from program outputs and other important parameters.

Reactive programming works well when employed in an imperative setting. The use of relations and the effective change in program behavior is high possibility in Java. In fact, any language that can directly control and describe hardware components such as Verilog can benefit from the use of reactive programming. It gives improved control over the available hardware resources.

Reactive programming works well by understanding that there is a publisher that keeps producing data, while there is also a subscriber that requires asynchronous access to the process information. This relationship is best described by Figure 19.1.

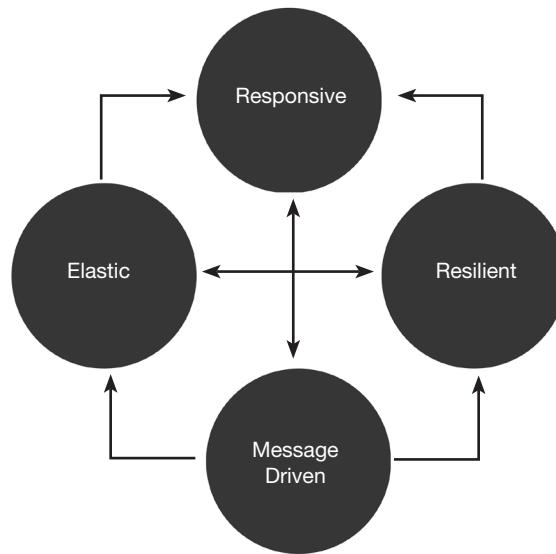


**Figure 19.1** Publisher and subscriber relationship for reactive programming.

There are several benefits to implementing reactive programming. Here are two common advantages that you get as a reactive programming expert:

1. You can use a simpler code for the required tasks. This allows you to create a readable code, which is easy to implement as well as improve when required in a client application.
2. It allows programmers to focus on implementing business logic in their programming solutions. This takes them away from following a conventional boilerplate code to stay away from similarity and come up with unique programming solutions.

There are four key attributes of a reactive system, which are described in the reactive manifesto. This is illustrated in Figure 19.2.



**Figure 19.2** Four key attributes of reactive systems.

1. **Responsive:** The system should respond in a timely manner, it not only ensures usability but also effectiveness.
2. **Resilient:** The system should respond in case of failure as well. This can be achieved by focusing on containment, replication, delegation, and isolation.
3. **Elastic:** The system should stay in elastic state. In other words, it should respond to varied workload. In the case of high workload, resource allocation should increase. In case of low workload, resources can be released.
4. **Message driven:** The system should ensure loose coupling, isolation, and location transparency by establishing boundaries between components and relying on asynchronous message passing.



What is the use of “reactivity” in Java?

Concurrency issues are better controlled with reactive programming techniques. The need for creating low-level threads that often depend on ideal synchronization is significantly reduced, thereby resulting in the creation of an environment which is conducive to efficient programming.

Reactive programming improves the efficiency associated with memory use. It is possible to create different streams that are then implemented with a multithreading mechanism to further enhance the results of employing reactive programming through the ideal APIs.

Reactive programming is a successful model with the ability to offer great results in all kinds of programming problems. It is not limited to only offer improvements in real-time applications, as it can be implemented to create reactive hardware definitions that use multithreading schemes.

This model is great for introducing powerful functions that use the available processors to carry out data transformations. There is no need to change either the subscriber or the publisher in the programming model. There can be an  $N$  number of processors that can work on incoming data streams and then provide results to the subscriber according to the particular needs of the application. This technique ensures that programming nodes can be made independent and have the capacity to handle any situation by simply changing the rules that work on the received data streams issued by the publisher.

Java 11 is specially designed to facilitate cloud applications that run in a real-time environment. This means that the language must provide resources that allow for a speedy process and ensure that it is possible to provide the best output, according to the dynamic inputs controlling the situation.

Reactive programs are interactive and are ideal to deal with the changing environment, which is the need in a data center providing support to cloud applications. However, reactive programming through Java interface also has several other applications. It certainly has the ability to create hardware drivers, provide a better virtual machine, and improve protocols that handle data streams.



How will you lose stream data in case of accessing it via concurrent program?

### 19.3 | Reactive Programming

Since Java 9 update, Java has great potential to carry out reactive programming. In fact, the reactive style is the best one that you can use in Java as it works well with conditions where you can declare the direction that the program must take in the case of receiving specific inputs and processing requests. Since Java 9 update, Java has Flow API that allows the use of reactive streams. This allows programmers to create and filter observable objects that can then be used to implement a dynamic behavior change, whenever it is required in any setting.

Spring is a popular Java framework in this regard, which can be employed for implementing strong reactive patterns and creating applications with the ability to show resilient behavior and improved performance. Throughout the course of this chapter, we will describe various resources and show relevant examples that will allow you to use several APIs to implement reactive programming that performs well in dynamic needs. These are the needs that are specifically required when working with cloud-based applications that need to use large heap sizes and perform better when there are increased processing needs.

The Flow APIs in JDK 9 and after are now working according to the Reactive Streams Specification. There are various implementations which support this standard, while the main goal remains the application of the reactive programming framework that was described earlier.

The API uses a model which depends on a push and a pull model. The Observer is a push mode, where source data items are pushed to reach the application. On the other hand, the pull comes from the Iterator in the Flow API, where the application actively pulls items present at the data source. The API runs by first requesting  $N$  data items and the publisher then pushes a maximum of these  $N$  items to the subscriber. The forwarded items may be less if required. The Flow API therefore performs by carrying out a mixture of pull and push steps for reactive functionality. Here is an example of how this API may function, which was taken from the Oracle Community Site (<https://community.oracle.com/docs/DOC-1006738>):

```

public static interface Flow.Publisher<T> {
    public void subscribe(Flow.Subscriber<? super T> subscriber);
}
public static interface Flow.Subscriber<T> {
    public void onSubscribe(Flow.Subscription subscription);
    public void onNext(T item) ;
    public void onError(Throwable throwable) ;
    public void onComplete() ;
}
public static interface Flow.Subscription {
    public void request(long n);
    public void cancel() ;
}
public static interface Flow.Processor<T,R> extends Flow.Subscriber<T>,
Flow.Publisher<R> {
}

```

This describes the flow process of a typical Java API that employs reactive programming. It uses the pull and push phases that were discussed. We will further present how to employ the individual functionalities from the subscriber and the publisher that are present at the operating end of a Java code that uses reactive programming streams and principles.

**QUICK CHALLENGE**

Take the example of a bank ATM and write a program using the reactive programming concept which allows user to withdraw cash from any ATM.

## 19.4 | What is Multithreading?



In the context of computer processing, multithreading refers to the capability of using multiple execution threads that can occur independently of each other through a unified process. The threads use the same pool of resources but have different bits of information that require processing (including exception handles), the CPU register requirements, and the stacking status in the addressing space.

The use of multiple threads is an excellent approach for empowering processes on a single processor system. This allows the use of at least two threads where one can always be responding to the user, while another one may be in working condition. However, since most modern computers have multiple processors, the power to employ multiple threads creates an ideal concurrency solution.

Multithreading is also associated with the need to program in a careful manner. It can cause deadlock and conditions where the processor finds it difficult to make the ideal execution decisions. 64-bit operating systems such as Windows often employ pre-emptive multithreading where the software is responsible for switching between different threads. This allows for switching to a high priority thread while using a low priority thread as the trigger element. Another method is the use of cooperative multithreading, which is extremely powerful but creates deadlocks if there is any problem during the execution of a single thread.

Most multithreading occurs at a blistering pace, in which the available time slices into many pieces in the fraction of a second and queued for different threads. This gives the impression to the user that all the threads are running in parallel, when in actual reality, they are taking turns during the processing cycles which is too fast for a typical application.

The primary advantage of using this method is to efficiently employ the available computational resources. A single thread may not often employ the available cache in a physical system. With the presence of multiple threads, it is possible to use the available CPU resources more efficiently, because most of them depend on the results of the current execution.

Running different threads ensures that the resources do not remain idle and will be employed by one or the other thread from time to time, within the microsecond execution cycles. However, multiple threads may interfere with each other if their use is not carefully coordinated. They also face problems if used at lower frequencies. Modern computers have significantly reduced these problems and provide an ideal scenario in which multithreading can be employed with improved accuracy and less failures.

Hardware manufacturers such as Intel claim that the use of multithreading can cause a 30% improvement in the performance of their processors and related components. Processes that often require floating point operations may gain as much as double the performance with parallel processing.

This means that with the right exposure allowed to software elements, multithreading can significantly reduce the execution time of a computer program, allowing for swift processing. It improves user experience and provides important processing advantages with reduced stop-times during executions.

#### 19.4.1 Multithreading in Java

Java is often employed to create programs that may serve several users through a single supporting platform. The thread in Java can be best defined as the smallest processing unit which can be employed to implement multitasking in a program environment. All threads share common heap, thereby ensuring efficient use of the available memory. The switching occurs according to a well-defined context which always happens faster than the time required for the processing of a thread.

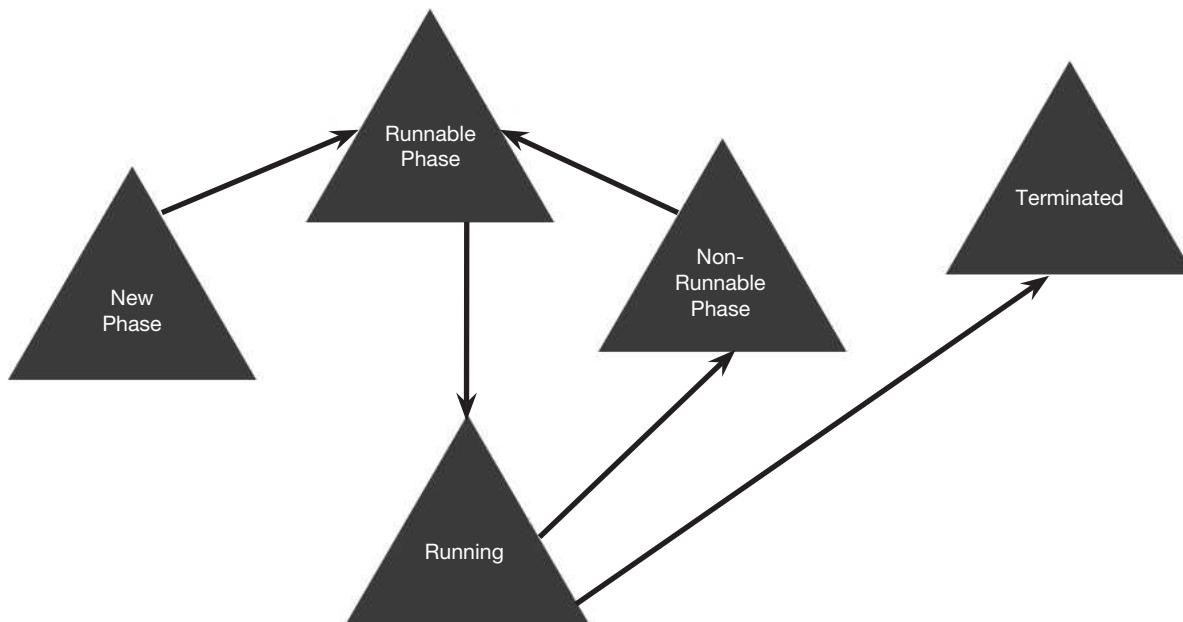
Multithreading in Java is an excellent way of improving the performance of programs that produce animations and complex decisions such as games. It is an excellent technique because it can produce the simultaneous results required in such programming environments. You get the following advantages when using multithreading in Java:

1. It is possible to run multiple operations in your program at once, thus reducing the overall time required to produce a response for the program user.
2. Threads are independent in terms of their program decision-making, which allows the user to always have an available interface.
3. Independent threads are excellent for handling program execution exceptions. Stopping one thread does not stop the entire execution and simply alters the resource allocation to produce excellent user interaction.

Understanding the life cycle of a Java thread is important. A thread goes through four separate phases (i.e., if we do not count the running phase), which actually does not affect the thread itself. The four phases are:

1. New phase.
2. Runnable phase.
3. Non-runnable phase.
4. Terminated.

Figure 19.3 is a visual presentation of how these phases interact with each other, including the actual running of the thread.



**Figure 19.3** Thread life cycle.

The new phase is the initial phase of any thread, which is created when the Java program calls for a Thread class instance. However, this is the phase of the project before the invoking of the `start()` method that produces the next phase.

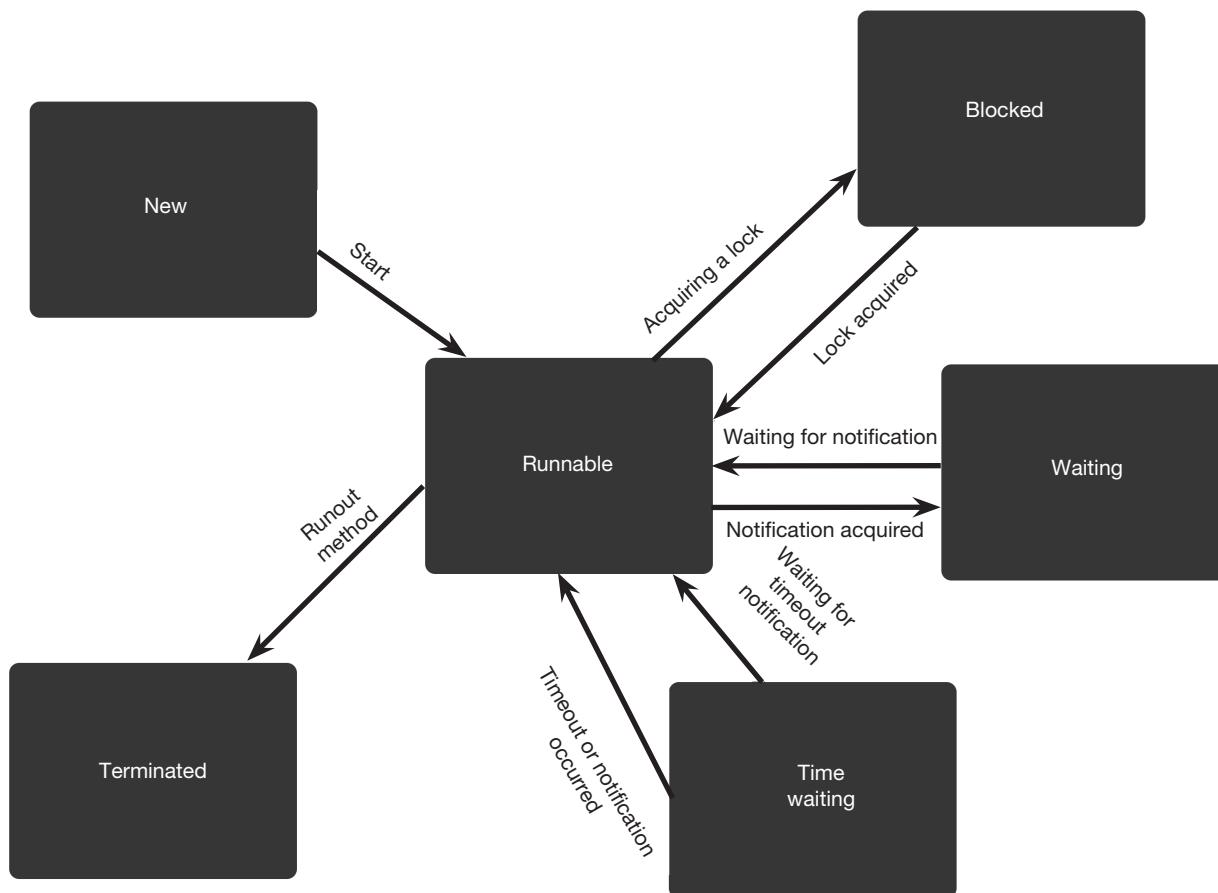
With the above method employed in the code, the thread is now ready for execution. This is termed as *runnable phase*. However, the actual running of the program is based on when the thread scheduler places the thread on a processing schedule.

The thread that enters the *non-runnable phase*, where it is blocked from further additions. This is important because the thread is still alive and any modification or further processing can cause program errors. The termination is identified when the `run()` method exists for the thread. This is also termed as *dead state*.

### QUICK CHALLENGE

Explain the thread life cycle concept using a real-life example of a bank ATM in which the user can withdraw cash from any ATM.

See Figure 19.4 to understand the thread states from a different view.



**Figure 19.4** Transition of threads from different phases.

### 19.4.2 Programming with Multithreading

A thread is created in Java by either extending the Thread class or starting the Runnable interface. The use of the Thread class provides the required methods and constructors that contain the operations that must be performed on the objects of the class. There are several constructors such as `Thread()`, `Thread(Runnable r, String name)` that are employed for creating threads.

Methods such as `join()`, `start()`, `run()`, `sleep()`, `getPriority()`, and `setPriority(int priority)` are employed in the Thread class objects. Other methods may include testing the thread status, returning the thread id, or even

modifying the name of the thread. The `start()` method initiates a new threat and sets it up in the calling stack. Once the thread is selected for processing, it always executes its `run()` method to perform the required functionality. Here is an example of a functioning Java thread:

```
package java11.fundamentals.chapter19;
public class JavaMultiThreadingExample extends Thread {
    public void run() {
        System.out.println("My newThread is Running");
    }
    public static void main(String args[]) {
        JavaMultiThreadingExample newThread = new JavaMultiThreadingExample();
        newThread.start();
    }
}
```

The above program produces the following result.

**My newThread is Running**

This is a simple thread that will print the message, “My newThread is Running”. It creates a single class instance with one `run()` method. This method extends the `Thread` class to carry out the intended functionality. Another way to perform the same functionality is to create runnable instances as presented in the following example:

```
package java11.fundamentals.chapter19;
public class JavaMultiThreadingWithRunnableExample implements Runnable {
    public void run() {
        System.out.println("My newThread is Running");
    }
    public static void main(String args[]) {
        JavaMultiThreadingWithRunnableExample myRunnableObj = new
JavaMultiThreadingWithRunnableExample();
        Thread newThread = new Thread(myRunnableObj);
        newThread.start();
    }
}
```

The above program produces the following result.

**My newThread is Running**

This one implements the `Runnable` interface. Since you are not extending, you need to show the creation of an explicit `Thread` class object, which is `th1` in this example. Another important concept is the thread scheduler. It is an important Java virtual machine (JVM) component that decides which of the runnable threads will be chosen next for execution. It can employ the method of time slicing or pre-emptive scheduling. Time slicing treats all threads equally while the pre-emptive scheduling allows for the setting of priority in the available threads.

In this regard, the `sleep()` method holds important value in Java. It pauses the thread for the mentioned milliseconds, which may be defined. The Java thread scheduler will ignore the thread and move on to the next one if it finds a selected thread with a running `sleep()` method. Here is an example of its use:

```

package java11.fundamentals.chapter19;
public class JavaMultiThreadingWithSleepExample extends Thread {
    public void run() {
        for (int i = 1; i < 4; i++) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
            System.out.println(i);
        }
    }
    public static void main(String args[]) {
        JavaMultiThreadingWithSleepExample myThread1 = new
JavaMultiThreadingWithSleepExample();
        JavaMultiThreadingWithSleepExample myThread2 = new
JavaMultiThreadingWithSleepExample();
        myThread1.start();
        myThread2.start();
    }
}

```

The above program produces the following result.

1
1
2
2
3
3

The sleep method throws an exception whose status can then be found with catch and then printed. There are two threads, and the running of each thread then produces a sleep period of 500ms. This means that myThread1 will get ignored in the next cycle and myThread2 will get picked up for processing. This will result in this program printing two 1s, two 2s, and two 3s as the console output.

All threads in Java can only run successfully just one time. If a thread is wrongly called multiple times, all other instances after the first one will return an illegal thread exception error. If the `run()` method is directly employed without first initiating the start, it produces the addition of the thread on the same call stack. The direct running of Thread objects is of no use, as they lose their functionality as individual threads and are simply treated as normal code objects.

Synchronization is also possible with the use of `join()` method. This is a method that stops a thread from executing until another referenced thread has been truncated. This is excellent for creating synchronized code, which may be important in several Java programs. You can also find out information about the thread which is currently running by using the `currentThread()` method. Changing the name is also possible, but it only holds value when this is used with other programming elements to produce the required functionality in a specific program.

The next important topic in this regard is to understand how the thread scheduler sets up the priority of the available threads. Java provides a priority value to every thread in a program. This is presented as a number ranging from 1 to 10. Normally, Java scheduler uses pre-emptive scheduling where the threads are arranged for execution according to these priority values. Although this may not be the case if a particular JVM is performing time slicing of multiple threads.

There are three important constants that are present in the Thread class. They describe the minimum, normal, and maximum priority for threads in the following forms:

1. `public static int MIN_PRIORITY`
2. `public static int NORM_PRIORITY`
3. `public static int MAX_PRIORITY`

The normal priority with a value of 5 is always selected for a thread as a default value. The `MIN_PRIORITY` directly sets the priority to 1 (lowest), while `MAX_PRIORITY` sets it up to 10 (highest).

An interesting point to note is that the JVM often creates a Daemon thread just to help the user created threads by providing them support and functionality benefits. Such a thread is automatically terminated by the JVM when all user threads are processed.



How will you guarantee thread priority?

Java also employs pool threads that are fixed thread objects that can be used for specific support actions. This allows for a quicker method, as creating user-defined Thread objects takes more processing time. You can also create multiple threads with a single object.

This is possible with the `ThreadGroup` class in Java. Multiple threads can be implemented within a single object of this class. All groups and their individual threads can be named in a customized manner. Here is an example that describes how the `ThreadGroup` may be set up. In an actual use, the described functionality may be complex, with each thread requiring specific processing power.

```
package java11.fundamentals.chapter19;
public class JavaMultiThreadingThreadGroupExample implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) {
        JavaMultiThreadingThreadGroupExample runnable = new
JavaMultiThreadingThreadGroupExample();
        ThreadGroup myThreadGroup = new ThreadGroup("My Thread Group");
        Thread t1 = new Thread(myThreadGroup, runnable, "My First Thread");
        t1.start();
        Thread t2 = new Thread(myThreadGroup, runnable, "My Second Thread");
        t2.start();
        Thread t3 = new Thread(myThreadGroup, runnable, "My Third Thread");
        t3.start();
        System.out.println("My Thread Group Name: " + myThreadGroup.getName());
        myThreadGroup.list();
    }
}
```

The above program produces the following result.

My First Thread
My Third Thread
My Thread Group Name: My Thread Group
My Second Thread
<code>java.lang.ThreadGroup[name=My Thread Group,maxpri=10]</code>
Thread[My Second Thread,5,My Thread Group]

This is a program that prints the name of the individual threads of the group as first, second, and third, and then the thread group name is returned as the Parent Thread Group after the first three threads are executed. The `list()` method then prints out the complete information of the created thread group `tg1`.



What will happen if two threads access the same resource at the same time?



## 19.5 | Concurrency

Concurrency is an important concept in computer programming. It defines the ability of a program to be executed in such a manner that any change in its running order does not affect how the final output is produced by it. It can include parallel processing of different processing units as well, which may be concurrent with each other and can be executed without affecting each other.

This improves the performance, especially in modern computers that have multiple processors with the ability to run multiple threads at the same time, which are independent of one another. The concurrency requires the decomposition of a program in partial elements so that these parts can be concurrently executed to use the available resources with improved efficiency.

Concurrency can certainly lower the program execution time and especially provide support in scenarios where an application may slow down due to a build-up of threads that may be run when performing sequential processing. All languages employ complex mathematics to create highly specialized and efficient concurrent processing schemes to improve program execution.

As the name suggests, the use of concurrency in computer programs results in various computations, which all overlap one another. The need for the sequence is eliminated, although a single program may use both sequential and concurrent program execution. Concurrency allows the application of modular programming, where the results from different computations can all be combined to produce the desired program functionality.

### 19.5.1 Advantages of Concurrency

The use of concurrency in Java programs can offer various benefits to programmers. Here are the top advantages that you can get with concurrent programming practices:

1. There is improvement in response of the programs. The program elements do not have to wait for specific input/output computation results, allowing the processor resources to be fully employed, while one thread is waiting for the results of a specific processing function.
2. There is improvement in the throughput of the program by a considerable margin. This is possible because parallel execution takes place constantly. This results in more tasks getting completed in every processing or execution cycle in the program. It can be simply described as a measure that shows improved resource efficiency.
3. There is improvement in the creation of better program structure. Concurrent programming is excellent for most problem domains that may require the result of several individual processes to finally reach conclusion. This may be the case where a program performs a complex function, which will often depend on the calculations of several individual elements that can be executed in any possible sequence.

Concurrent programming can use different methods. An ideal way to implement concurrent programming is multithreading. A set of threads allows an operating system or a JVM to use multiple processors to run parallel executions and therefore, gain the benefits of concurrency. Java is a language that uses explicit communication to describe the use of concurrent programming.

Java and C# are important programming languages that use communication during concurrent components that share memory resources. This is accomplished by setting up threads that can coordinate through locking mechanisms. A program that has the required functionality to avoid problems between parallel threads is termed as *thread-safe product*.

There are other methods, but we are not going to discuss them since they are not associated with the concurrency employed through specific APIs. Java implements the concept of concurrency with the use of the Thread class that we have explained above, as well as employing runnable instances which provide control over the execution of different threads.

### 19.5.2 Concurrency in Java

For typical computer users, they always expect that their computers are going to perform multiple tasks at the same time. They want their word processor to accept their keyboard inputs and display them on the screen, while still listening to songs that are directly streamed from the Internet. This functionality is only possible with the use of concurrent software.

Java has been supporting concurrency since its version 5.0, and the latest version is a lot more powerful. It does not provide the basic concurrency support in its JDK and class libraries, it also has an excellent collection of concurrency APIs that can perform at the highest level. The `java.util.concurrent` package contains powerful APIs that allows programmers to implement advanced concurrency and increase the throughput performance of their programs.

Java uses two units in concurrent programming: threads and processes. Most concurrent programming functions are delivered and performed on threads. Sometimes processes are also employed during concurrency assignments. A computer system often has a host of open processes and threads at any given time, regardless of the number of cores on the processor. Take the example of an operating system, where you can check the number of processes that are operational at the same time.

Understanding processes and threads in Java is ideal for understanding how concurrency works, especially in Java 9 update which has specific concurrency API improvements. A process is defined as a set of execution resources that have specific heap space. Processes are often defined as applications or programs, although it is possible that a single user application may be running multiple processes in the background.

The JVM mostly employs a single process, when using the computational resources of the hosting computer. However, Java provides support for creating multiple processes as well. It is possible with the use of a special `ProcessBuilder` object, which is a specific application beyond the scope of our concurrency article.

Threads, on the other hand, are small (lightweight) processes. They are simply termed as the most basic unit of execution that is delivered to the physical processor by JVM. Threads are also a part of the execution, but it is easier to prepare them as fewer resources are required. They always occur within a singular process in Java, while each process in any application would always have a single thread.

All threads share the resources that are allocated to the overall process in Java. This includes the heap which is assigned to the JVM as well as the open files, according to the libraries mentioned in the program bytecode during executions. Although this creates an efficient recipe, it is harder to control the use of resources without using an effective communication system for inter-thread messages.

The Java platform allows the use of multiple threads, as they are always present, even if your program only asks for one. The system also creates its own threads to perform important functions like built-in memory management. The programmer must only focus on the main thread, which is the one that contains program instructions.

Now, it is important to understand how the JVM can employ them in order to improve the performance of a Java program. Most Java applications can create multiple threads that may then be used for parallel processing or for implementing asynchronous behavior in the program.



Is there any alternative to concurrency? Explain.

Concurrency is the technique which ensures that all tasks can be speeded up by breaking them into smaller tasks that can occur independently of one another and executed in parallel in different threads. The performance of Java Runtime depends strongly on the results of the current parts that are getting processed. The Amdahl's Law governs the maximum performance gain that can be achieved with this practice as:

The  $F$  denotes the percentage of the program that must run in a synchronized manner and cannot go through parallel processing and  $N$  is the total number of processes that are running at any given time. Concurrency is not free from its own problems. This happens because threads can control their call stack, but share heap locations with each other. The first problem that concurrency produces is that of visibility while the second one is about the ideal access.

The first problem of visibility occurs when the data shared by the first thread is then used and changed by another thread without informing the first thread about it. This means that when the elements of the first thread go in their next processing, the change of data completely eliminates the functionality of the program and produces wrong data inputs within them.

The second problem of access occurs when multithreading and parallel processing is implemented during a concurrency run. This means that two different threads may attempt to access the shared data at the same time, while changing it after their particular processing in a single go. This creates a deadlock as the program cannot comply with multiple demands to access and change the same data location from two different threads at the same time.

The `java.util.concurrent` API package is important in this regard as it provides the support required for creating different threads and implementing strong measures for the required code synchronization. It is important that threads that are sharing data sources must run in an organized manner, where it is not possible to corrupt the data for the other thread.

This is possible with the code locks that Java implements. It ensures that several threads can run at the same time, but never use a similar call to data that can create deadlock situations. It is simply implementing with the use of `synchronized` keyword in Java. There are various benefits of using this technique in concurrency programming in the language:

1. It ensures that a singular code block is only employed by a single thread in the same timeslot. This ensures that the duplication and overwriting of data is not possible.
2. Each thread is able to view the previous modifications that occurred to the data objects in the code. This ensures that no incorrect data is processed, allowing for the removal of deadlocks and inaccurate situations.
3. It provides the blocks of mutually exclusive access to code elements. This is important as it allows threads to communicate with each other and always ensure that the shared data is current with the needs of the concurrency in Java programs.

The keyword can be easily used when defining any method in Java. This ensures that only a single thread will be able to use the code block then, during parallel execution of different program threads. The next thread that needs this code will wait until the first thread has used and left the locked method. Here is an example of its use:

```
public synchronized void myMethod()
{
    // thread critical information
    // the intended functionality
}
```

The synchronization can also be used with individual code elements that may actually be a part of a method within an object. This then creates a locked block, which is guarded by a key. The key can be created in the form of an object or a string value to create the intended lock, required for ensuring that no problems occur during concurrency. Here is an example that helps present the use of locking code sections and blocks for seamless concurrency:

```
package java11.fundamentals.chapter19;
import java.util.ArrayList;
import java.util.List;
public class SynchronizationExample {
    private List<String> wishList = new ArrayList<String>();
    private List<String> shoppingCart = new ArrayList<String>();
    public void addToWishList(String product) {
        synchronized (this) {
            if (!wishList.contains(product)) {
                wishList.add(product);
            }
        }
    }
    /**
     *
     * Now the code moves to add the next product to the shopping cart.
     * If there are no products left in the wish list, it returns Null.
     */
    public String addToShoppingCart() {
        if (wishList.size() == 0) {
            return null;
        }
        synchronized (this) {
            // Checking if any product is available in the wish list
            if (wishList.size() > 0) {
                String s = wishList.get(0);
                wishList.remove(0);
                shoppingCart.add(s);
                return s;
            }
            return null;
        }
    }
}
```

Java also provides another method to ensure that threads do not make a mistake when picking up values for the required fields. This is accomplished by using the *volatile* keyword in the declaration of the variable for any Java class. It guarantees that whenever a thread is using this attribute, it will read the most recent value for the required information. However, it does not create an exclusive lock on the variable. There are conditions where this mode of functionality is required in a program.

A volatile variable automatically updates the variables as well, if they are modified within a single thread. This means that such a variable can often work as a reference for multiple values that may get changed during a temporary case scenario. The

setter is employed in such settings to initialize and assign a value to the variable. This allows for placing an address change and allowing the stored values to be available for other threads that may attempt to use it.

Concurrency significantly depends on the accurate use of the available memory for the JDK and the JRE. The memory model controls the communication between the memory which is assigned to the individual threads and the main memory available to the entire Java program, which is allocated by the JVM when the program is running on a particular computer.

The memory model is responsible for creating and defining the rules that govern the use of memory by different threads. It also controls the way information is communicated between the different threads. It describes the solutions for keeping memory available for the program. This is produced by allowing a thread to update its use of memory from the available main memory.

Atomic operations are important in Java, as they are defined as standalone tasks that must be completed without any interference from other program tasks. It is important that atomic operations are identified during their execution. Java allows the specification of a variable, when it is running an atomic state. This is automatically possible, but operations with long or double literals must be defined as atomic by using the *volatile* keyword with their declaration.

Normal operations like increments and decrements using the `++` and `--` operators are not considered atomic by Java, when performed on integer datatypes. However, by defining the value as an atomic one, it is possible to use such functionality. Java 9 onwards, Java supports the use of atomic variables that are already defined and have the necessary methods to perform different mathematical operations.

The synchronization is carefully maintained by the memory model as it updates information when it finds that a block of code is performing modification on a locked set. All previous modifications are available to the model that ensures excellent integration and the ability to simply remove all deadlock instances.

The simplest way to avoid problems with concurrency is to share only immutable data between threads. Immutable data is data which cannot be changed.

### 19.5.3 Concurrency Support

Java understands the importance of supporting concurrency. It provides several support methods that ensure that a class cannot be changed by proving several elements. All the fields and the class declarations are immutable when they remain finalized and their reference does not move during the construction of the class.

All fields that describe movable data objects remain private and are not used with a setter method. They are also not returned directly. Even if their value is changed within the class, it does not affect their use outside of the class in any manner whatsoever. This means that it is possible to have a fixed class that contains objects that have mutable data.

Take the example of mutable information such as Arrays that may be actually handed to a class externally, when the class is in its construction phase. The class can protect such elements by creating defensive copies of the required elements. This ensures that an object outside of the class cannot change the data in such fields.

Defensive copies play an important role in protecting your classes. This is important because other code elements can call for the class and in turn, change the data present in it in a manner not anticipated in your program's logic. A defensive copy is an excellent mechanism for ensuring data integrity. It works in a simple manner. Whenever a calling code asks for information, the immutable class creates a data copy and provides it to the code, without ever affecting the actual information stored in the immutable class. Here is an example of a defensive copy:

```
package java11.fundamentals.chapter19;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class DefensiveCopyExample {
    List<String> myArrayList = new ArrayList<String>();
    public void add(String s) {
        myArrayList.add(s);
    }
    // Following code creates a defensive copy and return the same. In this case, the
    // returning list remains immutable.
    public List<String> getMyArrayList() {
        return Collections.unmodifiableList(myArrayList);
    }
}
```

This program creates a new list for as a copy of the original list and passes it on to the relevant code. The original list never changes its status as array values.

Another important concept is a thread pool that holds the work queue for a Java program.

The pool keeps a record of all Runnable objects and constantly updates the list as and when required by the program. You can use the `execute(Runnable r)` code if you want the current thread to enter the queue where it will be called according to its preference that was explained earlier. The pool is named Executor and its implementation is utilized from `java.util.concurrent.Executor` interface. Worker threads can be easily added to the Executor, while there are also methods available to shut it down and terminate the thread processing.

Java can handle all types of asynchronous operations with the availability of this particular interface. There are different ways to implement execution tasks, such as using the Future interface. It allows asking for the results from a Callable task in Java. The CompleteableFuture option is available since Java 9 is very strong, as it ensures that all time-consuming activities are arranged in an ideal manner.

It allows the use of two approaches to provide concurrency in Java programs. The first is to ensure that all application blocks are arranged in a logical manner and follow the steps that are required to complete particular tasks. Another is to create a non-blocking approach where the application logic only moves with the flow of the tasks that are required for a program. This functionality allows the creation of different steps and stages and provides better control over the code callbacks that are required in any Java application.

#### 19.5.4 Concurrency API Improvements

Java 9 contains considerable concurrency API improvements that occur as defined in the JEP 266 document. Here, we describe the improvements that are shared by the Java 9 documentation regarding the ability to better use `CompletableFuture` class and providing a lot more power in the language when compared with the older version.

The main focus of the concurrency improvements is the `CompletableFuture` API. The motivation behind these improvement steps is that the concepts of concurrency and parallelism must be fully integrated in the programming and execution requirement to give optimum control to a Java developer.

The improvements appear by providing better support in the relevant Java libraries, in turn adding the added functionality to the classes and the methods that may come under them, especially related to threads and concurrency settings. The interface improvements are also based on creating Reactive Streams that use the principles of concurrency and parallel thread processing.

Reactive Streams can provide better support through the use of the class `Flow`. It allows publishers to create items for different subscribers. The individual solutions can all use simple communication with the use of an information flow control to provide the ability to react to a dynamic program execution environment. A utility class is also added in the form of `SubmissionPublisher` that allows developers to create customized components capable of independent communication.

The main enhancement remains on the `CompletableFuture` API that was already discussed. Time-specific enhancements were certainly required during concurrency operations in Java as they had the ability to ensure consistency and make sure that all deadlock situations can be eliminated with accuracy.

There are excellent methods that are added to control the duration of different threads and their relevant operations. An Executor is added, which is extremely powerful and provides the use of a static method, such as `delayedExecutor`. It is a powerful method with the capability of providing a time gap between the reading of the code and the execution of a particular task.

When combined with the Future functionality, it can create excellent support for complete threads that can use concurrency, but still provide the reactivity necessary for enjoying the ideal support for improved parallel processing. There are several small improvements as well, which may not be apparent to a normal programmer, but significantly improve the capacity of handling multithreading that ensures reactive programming. This is built within the concurrency structure to offer improved throughput while maintaining greater control over enhanced heap sizes and shared memory spaces.

The overall concurrency improvements create a significant difference between the reactive programming capability of Java 8 and Java 9. The new programming kit employs the Flow API at its maximum power by providing four static interfaces. These interfaces contain all the methods that can provide a flow in program executions, where the publisher can control the production of data objects according to their consumption by different consumers of the program.

The production is handled by a Publisher, which produces the data required and received by different subscribers. The Subscriber acts as a receiver, where its argument depends on the definition presented by Subscription, that links it with the Publisher. Both parties combine together to create the Processor, which specifies a specific transformation requirement.

The data streams can be set up with the use of the Publisher by using the `subscribe()` method. It is used to connect a Subscriber with a Publisher. If a Subscriber is registered already, then this method registers an error and returns an illegal exception.

A Subscriber works to provide callback code for data items through the Flow library. It can be declared with `onSubscribe()` method. However, the same declaration can also occur with `onError(Throws exception)`, `onComplete()`, and `onNext(item)`. The first method describes a registration from allowing the requests for new data items to be moved to the subscriber. This situation is required for implementing the program flow measures that provide concurrent behavior.

There are also new methods that Java 9 update brings in the `CompletableFuture`. They allow the creation of different stages when performing concurrency functions. This ensures that the program always remains protected from sudden failures. A method file `completedStage()` and `failedStage()` are perfect for providing information that are in the processing phase. They can throw exceptions and provide information that may be useful for showing that a particular function is completed, ensuring the use of aggressive concurrency practices in the program.

The `CompletableFuture` is now more powerful, especially due to the support of delay and timeouts which are ideal for use in a large and complex program that often employs multiple threads and thread groups. The delay is an excellent choice for use in concurrent applications, where it simply associates the required time that significantly covers a thread from corrupting the available data values.

There is another method called `Timeout()`. It is excellent for ensuring that a particular future situation is eliminated before a relevant code must run. This situation may not always be required and therefore, the method only throws an exception if it is required for a `CompletableFuture<>` functionality. However, this functionality can be further improved by experimenting around and creating different logical approaches to achieve the desired functions.

Here is an important example of how the delay can be presented using different Flow controls:

```
package java11.fundamentals.chapter19;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
public class DelayedExecutorExample {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        CompletableFuture<String> completableFuture = new CompletableFuture<>();
        completableFuture.completeAsync(() -> {
            try {
                System.out.println("CompletableFuture - Executing the code block");
                return "CompletableFuture completeAsync executed successfully";
            } catch (Throwable e) {
                return "In catch block";
            }
        }, CompletableFuture.delayedExecutor(3, TimeUnit.SECONDS))
            .thenAccept(result -> System.out.println("In Accept: " + result));
        for (int i = 1; i <= 10; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Executing For Loop Block : " + i + " s");
        }
    }
}
```

The above program produces the following output.

```

CompletableFuture - Executing the code block
In Accept: CompletableFuture completeAsync executed successfully
Executing For Loop Block : 1 s
Executing For Loop Block : 2 s
Executing For Loop Block : 3 s
Executing For Loop Block : 4 s
Executing For Loop Block : 5 s
Executing For Loop Block : 6 s
Executing For Loop Block : 7 s
Executing For Loop Block : 8 s
Executing For Loop Block : 9 s
Executing For Loop Block : 10 s

```

This is a simple program that will present the running outside three times with values of 1, 2, and 3 seconds. The inside future will represent the screen output of processing data while presenting the required acceptance. The delay will then be overproducing the remaining 4, 5, 6, 7, 8, 9, and 10 seconds' values.

Methods that were not used in the previous version are removed and new methods are added in Java 9, which provide excellent support to the ever-improving concurrency API present in the programming environment. It includes improvement in the atomic functions that often employ reference array methods and Boolean comparison functions.

One problem that may appear with the Flow API is that sometimes, Publishers can produce data at a much faster rate, compared to its consumption by multiple Subscribers. These situations require the creation of an ample buffer, which can hold unprocessed elements. These elements produce backpressure, which Java 9 handles with the creation of logic that removes the collection of elements with various reactive programming techniques.

Reactive programming, when introduced with multithreading, truly empowers Java 9 and allows developers to use it at the best of their capacity. The development environment is still evolving to produce more API improvements. With the main improvement available in the Java package, it is inevitable that different developers may end producing better API support models that will help in using reactive programming principles with improved concurrency.

A conclusion to the concurrency API improvements focuses on describing how the use of effective evolution is required for using stronger techniques and methods such as multithreading. With improvement over the control of functions that are related to one another, it is possible to use the full advantages of concurrency as governed by its theoretical limit.

With the tools of CompletableFuture and new improvements like ForkJoinPool and other relevant classes, it is important to understand the individual threads in Java and how they can be set up to ensure the best use of multithreading and the ideal parallel processing with the help of parallel programming principles.

There are other API improvements as well. One important class in this regard is the ConcurrentHashMap, which is designed to support a system of concurrency retrievals. It is an excellent design capable of providing the Hash table functionality in an improved manner. It has the same methods, but all with improved performance. It is a class that does not employ locking, but offers all functions to remain safe from thread deadlock issues and other parallel processing problems.

The mapping class works well with retrieval and ideally allows the use of updating within the same processing zone due to not locking the code or data. Different parameters of the hash table can also be retrieved, allowing the use of enumeration or the iterator. The hash table has improved control and can also be resized if it faces a certain load. Size estimates are possible as well, by using the initialCapacity constructor. It is certainly a class that allows for improved concurrency in Java 9 when compared with the older JDK environments.

The newKeySet() method is available for setting the mapped values or simply recording the different positions. There are few differences from other classes, as it does not allow the use of null value. Concurrent hash maps are excellent for use when combining serial and parallel operations. They offer safe way to apply concurrency, while still ensuring that the ideal arguments can be used in the Java program.

ConcurrentHashMap generates a frequency map as well where it can produce a histogram of usable values. This means that it can support functions both in an arranged manner, while still containing a set of parallel executions that may be controlled with a single forEach action. Remember, the elements should always be used in a manner as to not get affected if the order of the supplied functions is changed since it will remain random, during bulk operations.

There are other operations of mapped reductions and plain reductions in this class. Plain reductions do not correspond to a return type, while mapped reductions are used for accumulating the application of functions. Sequential methods occur if the

map returns a lower size than the given threshold. The use of `Long.MAX_VALUE` provides suppression control on parallelism, while the use of 1 acts as allowing for maximum parallel results. This is possible because the program creates subtasks by using the `ForkJoinPool.commonPool()` method that allows parallel computations. The ideal programming starts by picking one of these extremes and then revising the values to achieve your required level of overheads against the delivered throughput.

The hash map can speed up the executions using parallel processing, but it is not always the preferred solution when compared to sequential processing. If small functions that are placed on separate maps are used, they will often execute slower than serial transformations. Parallel processing will also not be valuable if it is simply taking care of different tasks that are not related to each other, and do not produce a net gain over the normal capacity of the Java compiler.

The `ConcurrentHashMap` can also be created in the image of another map. This is an excellent option when you are experimenting with their use and have not yet identified the ideal approach to use the available `HashMap` options.

There are some excellent parallelism options since Java 9, such as the ability to use the `newWorkStealingPool()` method. This is a method present in the `Executor` class and allows the creation of a threading pool. It uses the available processes as the value for parallelism, and this defines the use of a process where all applicable processors are working simultaneously on different tasks. If the parallelism value is inserted in this method, then the thread pool keeps the required number of threads for the parallelism. The class then uses the creation of different queues to ensure that multiple threads are not in contention for the limited execution slots.

### 19.5.5 Dealing with Individual Threads

Let us once again discuss the individual threads produced by the JVM. Normally, the `main()` method creates a single thread on JVM. The thread continues to perform until the exit method of runtime class is called. The program ends when all non-daemon threads have performed their functions and have already been recalled. Another way for individual thread elimination is through an exception.

Individual threads can have several important parameters. It includes the name, the target, and the stack size available to the thread class object. The creation of individual threads is required especially when you want to implement the strong power associated with multithreading capacity of Java compiler.

There is excellent control available over the individual threads, with the `currentThread()` method allowing you to reference the currently processed object. There are other methods available as well, such as `yield()` which ensures that the thread is willing to drop its current use of processing for any other thread that is present in the processing scheduler.

The individual threads can also be controlled by momentarily making the caller unavailable for processing. Invoking a method like `onSpinWait()` is ideal for situations where you may want a loop construct that needs to show that the calling thread is, in fact, busy waiting for information from other parts of the program. This method keeps spinning unless controlled by a particular process or exception produced using the available flags. Here is an example:

```
package java11.fundamentals.chapter19;
public class OnSpinWaitExample {

    volatile boolean notificationAlert = true;

    public static void main(String args[]) {
        OnSpinWaitExample onSpinObj = new OnSpinWaitExample();
        onSpinObj.waitForEventAndHandleIt();
    }

    public void waitForEventAndHandleIt() {
        while ( notificationAlert ) {
            java.lang.Thread.onSpinWait();
            System.out.println("In While Loop");
        }
        processEvent();
    }

    public void processEvent() {
        System.out.println("In Process Event");
    }
}
```

The above example produces the following result.

```
In While Loop
```

The control over the individual threads is still available if they are grouped together. You can get the count of the threads, as well as learn about currently active threads. However, remember that it is possible to perform multithreading by allowing them to process in a concurrent manner, where they do not cause an interruption because of the shared memory space.

Another important concept to understand about individual threads is their ability to perform communication, which is termed as co-operation in Java. It is a method of allowing one thread to pause while ensuring that another thread can be executed in a particular order. It is accomplished using `wait()`, `notify()`, and `notifyAll()` methods that belong to the Object class in Java.

Let us start with `wait()` as it is a method that causes the currently operational thread to release its data lock and go into the waiting mode. The waiting can be for a defined period or only returned with the use of the notify methods. A time period can be mentioned for the return as the method argument or left without use for the following methods.

- notify():** This method causes a single thread to come out of the waiting stage and become active on the current object. If there are multiple threads that are present within a single object, the selection of the awakening thread is random and lies at the discretion of the system components.
- notifyall():** This method is excellent when you want all threads that are present in a particular object to come out of their waiting stage. This is important since it removes the random nature of the previous method and allows for better concurrency functions.

Remember, the `wait()` method is different from `sleep()`. The former method applies on the Object class while the latter works on the Thread class objects. The `wait()` method remains a non-static method and can be revoked with the use of `notify()` or `notifyall()` methods, as well as specific time. On the other hand, `sleep()` works with a specific amount of time, and does not provide the dynamic control which is required when actively performing aggressive reactive programming that takes full advantage of the capabilities of the Java compiler.

### 19.5.6 Synchronizing Code Blocks

Synchronizing code blocks is an excellent way of creating particular methods and then controlling them to work in an organized manner. Synchronization can also occur in a limited capacity, where we can place some part of a method within a synchronized block while leaving the rest of it for random execution.

It is in fact, a way of locking parts of code that you do not want to be accessible to any other resource. Remember, the scope of using a synchronized block is always smaller than a complete method and should always be employed in this manner. Here is an example of a synchronized block used within a Java program:

```
package java11.fundamentals.chapter19;
public class SynchronizationBlockExample {
    public static void main(String args[]) {
        Calculator calculator = new Calculator();
        WorkerThread1 t1 = new WorkerThread1(calculator);
        WorkerThread2 t2 = new WorkerThread2(calculator);
        t1.start();
        t2.start();
    }
}
class Calculator {
    void multiplicationTable(int n) {
        // Following block will ensure the method is accessible in synchronized manner
        synchronized (this) {
            for (int i = 1; i <= 10; i++) {
                System.out.println(Thread.currentThread().getName() + " : " + n * i);
                try {
                    Thread.sleep(400);
                } catch (Exception e) {
                    System.out.println(e);
                }
            }
        }
    }
}
class WorkerThread1 extends Thread {
    Calculator t;
    WorkerThread1(Calculator t) {
        this.t = t;
        this.setName("Worker Thread 1");
    }
    public void run() {
        t.multiplicationTable(3);
    }
}
class WorkerThread2 extends Thread {
    Calculator t;
    WorkerThread2(Calculator t) {
        this.t = t;
        this.setName("Worker Thread 2");
    }
    public void run() {
        t.multiplicationTable(40);
    }
}
```

The above program produces the following result.

```

Worker Thread 1 : 3
Worker Thread 1 : 6
Worker Thread 1 : 9
Worker Thread 1 : 12
Worker Thread 1 : 15
Worker Thread 1 : 18
Worker Thread 1 : 21
Worker Thread 1 : 24
Worker Thread 1 : 27
Worker Thread 1 : 30
Worker Thread 2 : 40
Worker Thread 2 : 80
Worker Thread 2 : 120
Worker Thread 2 : 160
Worker Thread 2 : 200
Worker Thread 2 : 240
Worker Thread 2 : 280
Worker Thread 2 : 320
Worker Thread 2 : 360
Worker Thread 2 : 400

```

This program uses a synchronized section within a single method of `multiplicationTable()` which includes a counter that then produces a sleep delay for the thread. This situation results in generating a set of numbers where multiples of 3 are printed 10 times, while the same is then repeated with multiples of 40. Similar functionality can be obtained by using an anonymous class where you do not have to define it separately for the program operations.

There are static methods as well, which only use fixed information. Using synchronization for such methods ends up locking the entire class, rather than a particular object that calls for the method. Take the example of two objects from the same class that share information.

They may be termed as `ob1` and `ob2`. The use of synchronized methods will ensure that interference is not possible between different actions. The use of static synchronization is excellent because it ensures that the lock is available for the class and away from the individual objects. The lock is easily created on the class by using this method:

```

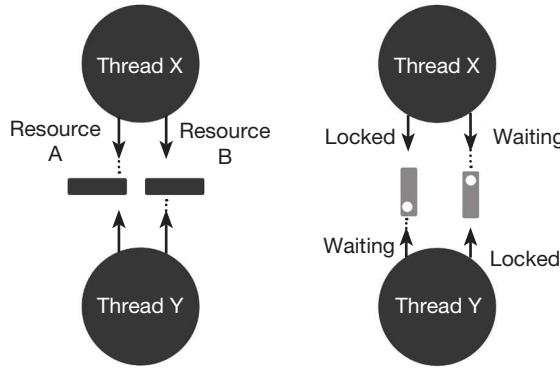
static void multiplicationTable(int n) {
    // Following block will ensure the method is accessible in synchronized manner
    synchronized (this) {
        for (int i = 1; i <= 10; i++) {
            System.out.println(Thread.currentThread().getName() + " : " + n * i);
            try {
                Thread.sleep(400);
            } catch (Exception e) {
                System.out.println(e);
            }
        }
    }
}

```

This is the declaration which can be available on the class when used with the static keyword at the top with a defined class method.

## 19.6 | Understanding Deadlock

Java 9 update brings excellent multithreading support, which certainly brings deadlock into the discussion. Remember, a deadlock is an unavoidable reality when performing multithreading. It is the inevitable result of a situation where threads must wait for locked objects and work according to the defined thread rules. Figure 19.5 shows a visual representation of deadlock.



**Figure 19.5** Visual representation of deadlock.

The deadlock is a natural situation that occurs in multithreading, when multiple threads are waiting for each other to release the lock on the object that needs to be processed, as the following example elaborates:

```

package java11.fundamentals.chapter19;
public class DeadlockExample {
    public static void main(String[] args) {
        final String firstResource = "First Resource";
        final String secondResource = "Second Resource";
        // Following code demonstrates thread 1 attempt to lock firstResource then
        secondResource
        Thread thread1 = new Thread("First Thread") {
            public void run() {
                synchronized (firstResource) {
                    System.out.println(this.getName() + " : First Resource is Locked");
                    try {
                        Thread.sleep(100);
                    } catch (Exception e) {
                    }
                synchronized (secondResource) {
                    System.out.println("Second Resource is Locked");
                }
            }
        };
        // Following code demonstrates thread 2 attempt to lock secondResource then
        firstResource
        Thread thread2 = new Thread("Second Thread") {
            public void run() {
                synchronized (secondResource) {
                    System.out.println(this.getName() + " : Second Resource is Locked");
                    try {
                        Thread.sleep(100);
                    } catch (Exception e) {
                    }
                synchronized (firstResource) {
                    System.out.println("First Resource is Locked");
                }
            }
        };
        thread1.start();
        thread2.start();
    }
}

```

The above code produces the following result.

**Second Thread : Second Resource is Locked  
First Thread : First Resource is Locked**

This is an excellent demonstration of the deadlock situation. The first synchronization in thread 1 already sets first resource to locked, which is then suspended by using sleep. This means that the program proceeds to run thread 2, which produces the screen output about second resource, while once again sleeping to allow thread 1 to run.

However, the program is now stuck due to deadlock because thread 1 must wait for second resource to be free of lock in thread 2, which means that the next thread for processing is thread 2. The thread 2 faces a similar situation as first resource is similarly locked by the thread 1, creating a condition where the program is logically stuck.

### 19.6.1 Resolving Deadlock

Many programmers choose to simply ignore that it is possible to have a deadlock condition during the execution of a Java program. They believe that a deadlock may only appear due to a poor programming effort, and should always be removed during the planning phase of the program. They also believe since a deadlock can freeze a program, a simple solution would be to restart the application, resulting in a new state where the same deadlock condition may never appear again.

Obviously, this is the wrong way to go about Java programming as you must ensure from your end that a program is free from any conditions that may produce a stall or freezing, especially if it is providing control and support for a large-scale operation. Another way to go about it is to build a detection system in your program.

This is achieved by adding a special task that gets executed only to check the current status of the program parameters. This ensures that it is possible to detect, if the program is entering a deadlock situation, due to the reasons that we just described above. This is possible by checking whether tasks are stuck in their current functionality. The remedy can occur by eliminating a stuck task or forcefully liberating a shared resource, which is required for other program elements to function normally.

Another method is to produce a prevention of the Coffman conditions. These are the four ways in which a deadlock can occur during the program execution. A program can prevent a condition where special measures are built within the program structure that stop the occurrence of these four situations.

Another method to avoid deadlock situation is to make sure that your program design avoids deadlocks. This is possible by ensuring that your program first obtains the information about the required shared sources each time a particular task starts in your program. This ensures that there is always a set of available resources that allow your task to get executed without any problem. If there is a lack of resources, you can insert conditional delays that will always ensure that only those starts are initiated that can be completed in the current set of available resources.

## 19.7 | Concurrent Data Structures



Java 9 update brings excellent data structure options, but they are never designed to provide the ideal support for concurrent operations. The use of an external method ensures that ideal synchronization is possible. However, it significantly increases the computing time of your application. There are certain data structure practices that you can employ that will allow you to create data structures that are fully supported by Concurrency API. There are two groups of these structures – blocking and non-blocking data structures.

The blocking structures are present as methods that ensure that the calling task is blocked when you are employing the data structure although it does not have the intended values. On the other hand, non-blocking structures do not block the calling tasks. These methods are designed to throw an exception if a task calls on the data structure when it does not hold value. It can also produce a null value as a result.

## 19.8 | Multithreading Examples

Multithreading Java examples can be best followed by understanding the thread pool in Java. It is a collection of Runnable objects. Worker threads in it are controlled by the Executor framework and create a life cycle where it maintains the overall life of the Java program. It is possible to use:

```
Executors.newSingleThreadExecutor()
```

This is a method that will create a single thread, but one which contains multiple running items. Here is how it can be applied:

```
package java11.fundamentals.chapter19;
public class RunnableExample implements Runnable {
    private final long counter;
    RunnableExample(long counter) {
        this.counter = counter;
    }
    @Override
    public void run() {
        long total = 0;
        for (long i = 1; i < counter; i++) {
            total += i;
        }
        System.out.println(total);
    }
}
```

Once the thread is established, it can be executed in numerous ways to gain the required facilities. We have significantly discussed the concept of Future. Now we describe that the Callable object can also be employed with concurrent processing. When this is used with an executor, it returns an object of the Future type. The `get()` method can be employed to receive a Future object which can then be employed for designing the ideal concurrent processing schemes.

An excellent use of thread control is possible when it is mixed with the ideal use of the `sleep()` method. Defining subclass is another effective method when implementing new threads. These are instances where you can control individual thread instances to gain better performance.

Another key way in which multithreading is performed is by using the Swing application. This is created through a set of conditions where there is an initial thread that includes the `main()` method, which is designed to exit at the occurrence of a defined event. This situation then gives rise to another event dispatching thread (EDT), which runs the specific functionality required from the program.

This technique provides excellent control over a program which may perform better with concurrent behavior. Multithreading is excellent when it can be divided in the form of different tasks that can then be set up according to the occurrence of key events during program execution. The main program then becomes a controller which keeps the operations shifting to the required threads. Remember, there is always the need to have a background (daemon) thread available, which takes care of the intensive tasks and the input/output controls that you need in a program environment.

When threads are switched according to their functionality, it is possible that application users may identify a pause between their inputs and the response produced by the application. This means that the EDT, which is set up in the program must never work for over 100 milliseconds, where it may start to produce a noticeable delay in the functioning of the worker threads.

Multithreading problems can be avoided in Java programs when the problems that are required for controlling the graphical user interface (GUI) are also performed on the EDT. This ensures that all operations remain thread safe, and can be carried out without ever causing any problems.

On the other hand, tasks that include the input/output processes should be kept on the main thread as they need to provide swift interaction, otherwise the program behavior is slow and does not produce the intended benefits. Another capacity for use is that Swing operations can be updated with the use of a delay timer. This allows the thread to update its components at controlled time intervals, resulting in the generation of the required information.

Another capability is with the publishing of thread information. This can happen with the use of:

```
protected final void publish(V... chunks)
```

The above method, which should be called inside the `doInBackground()` method, sends data to the process method to deliver intermediate results

```
protected void process(List<V> chunks)
```

The above method works in an asynchronous manner to receive datasets from the `publish()` method.

This is a program that produces the intermediate results. The method can be useful for implementing improved controls in the program. It allows Java programs to produce the situation of the current tasks. This allows for the implementation of practices that result in a better control over the data streams that are available in the program.

### 19.8.1 Matrix Multiplication

A common problem which you can use to learn concurrent programming is to create a matrix multiplication program. You can learn the simple concept of matrices where it is possible to multiply two matrices, as long as the number of columns of the first matrix is equal to the number of rows of the second matrix. This is termed as having matrices  $A(p \times q)$  and  $B(q \times r)$  where the first value in the parentheses depicts rows and the second value describes columns.

This operation is possible using different schemes. The Java library recognizes this need as an important function and provides the `MatrixGeneratorExample` class for these operations. This class uses the following code for its operation:

```
package java11.fundamentals.chapter19;
import java.util.Arrays;
import java.util.Random;
public class MatrixGeneratorExample {

    public static void main(String args[]) {
        System.out.println(Arrays.deepToString(generateMatrix(3, 3)));
    }

    public static int[][] generateMatrix(int rows, int columns) {
        int[][] matrix = new int[rows][columns];
        Random random = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                matrix[i][j] = random.nextInt() * 10;
            }
        }
        return matrix;
    }
}
```

The above program produces the following result:

```
[[853339708, 1266531654, -21594734], [1024418966, -490434492, -2020854294], [1604225446, -2030454084, 568464094]]
```

As we can understand from the structure of this class, it is easy to perform this calculation using a serial method. This will be a method where we will use two matrices and then use loops to multiply the elements that are present in the rows and columns

of the matrices. The results will be updated each time when calculating the value, and the loop will end when all elements are multiplied.

You should always produce a sequential version of every algorithm before you employ parallel versions. Here is the sequential method for multiplying matrices:

```

package java11.fundamentals.chapter19;
import java.util.Arrays;
import java.util.Random;
public class MatrixSerialMultiplierExample {

    public static void main(String args[]) {
        int[][] firstMatrix = generateMatrix(3,3);
        int[][] secondMatrix = generateMatrix(3,3);

        System.out.println(Arrays.deepToString(multiplyMatrix(firstMatrix, secondMatrix)));
    }

    public static int[][] generateMatrix(int rows, int columns) {
        int[][] matrix = new int[rows][columns];
        Random random = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                matrix[i][j] = random.nextInt() * 10;
            }
        }
        return matrix;
    }

    public static int[][] multiplyMatrix(int[][] matrix1, int[][] matrix2) {

        int row1 = matrix1.length;
        int column1 = matrix1[0].length;
        int column2 = matrix2[0].length;
        int[][] result = new int[row1][column1];

        for (int i = 0; i < row1; i++) {
            for (int j = 0; j < column2; j++) {
                result[i][j] = 0;
                for (int k = 0; k < column1; k++) {
                    result[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }
        return result;
    }
}

```

The above program produces the following result.

[[ -59142824, -631908344, -1314105512], [-341256416, 293358376, -1805709024], [194502204, -108105864, 1115015312]]
--

This is a serial program that uses three matrices. The first two matrices are multiplied using the basic matrix multiplication rules and the results are stored in the `result[][]` matrix. This program implies that both matrices are fit for performing the mathematical operation. This can be checked using other programming elements. We can also check the execution time by using a random generator to multiply large enough matrices, like ones with 2000 rows and columns, to get how many milliseconds it takes to reach the result. Here is a small section which shows how you can record time for the matrix multiplication:

```

package java11.fundamentals.chapter19;
import java.util.Arrays;
import java.util.Date;
import java.util.Random;
public class MatrixSerialMultiplierWithTimeExample {

    public static void main(String args[]) {
        int[][] firstMatrix = generateMatrix(3,3);
        int[][] secondMatrix = generateMatrix(3,3);

        Date start=new Date();

        System.out.println(Arrays.deepToString(multiplyMatrix(firstMatrix, secondMatrix)));

        Date end=new Date();

        System.out.printf("Total Time Taken : %d milli seconds", end.getTime() - start.
get Time());
    }

    public static int[][] generateMatrix(int rows, int columns) {
        int[][] matrix = new int[rows][columns];
        Random random = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                matrix[i][j] = random.nextInt() * 10;
            }
        }
        return matrix;
    }

    public static int[][] multiplyMatrix(int[][] matrix1, int[][] matrix2) {

        int row1 = matrix1.length;
        int column1 = matrix1[0].length;
        int column2 = matrix2[0].length;
        int[][] result = new int[row1][column1];

        for (int i = 0; i < row1; i++) {
            for (int j = 0; j < column2; j++) {
                result[i][j] = 0;
                for (int k = 0; k < column1; k++) {
                    result[i][j] += matrix1[i][k] * matrix2[k][j];
                }
            }
        }
        return result;
    }
}

```

The above program produces the following result.

```
[ [676643992, 1091729664, 1847180544], [1048256660, -2143242596, -332108212], [-1842184312, -1674877996, 974596884] ]
Total Time Taken : 1 milli seconds
```

This section of the code will first record the time before the start of the matrix multiplication and then also record the time when the final calculation has occurred. The output to the console is done by subtracting both times; this results in the number of milliseconds it took to complete the operation.

Now that we have learned how to carry out this program, it is time to prepare concurrent versions to see the difference. Sequential programming often has only one avenue, but there are various methods for parallel processing. You can use a single thread for each element when creating the result matrix. You can also employ an individual thread for every row for the same matrix. Another effective way of concurrent programming is to employ all the threads that are available in the current JVM structure.

The first method creates too many threads, as the number of elements are too great for large matrices. Take the example of  $2000 \times 2000$  matrices. Multiplying these matrices will result in the form of a matrix that will require the calculation of 4,000,000 elements. Although this can be accomplished, we will not be executing this program in this book. However, we will represent the other two options here.

## 19.9 | Designing Concurrent Java Programs



There are several ways to create a concurrent Java program, one that employs the best principles of parallelism, while using the different resources and tools that are present, especially in the latest Java development environment. Here is a strategy that you can use to ensure that you always end up with the required concurrent algorithms for use, regardless of the actual functionality that you need in the program.

A good practice in this regard is to always create a sequential version of the algorithm that you want. This is important as it allows you to measure the advantages of using concurrent code, as well as understand whether both versions are producing the same program results, which is important in all applications.

The creation of an initial sequential program will allow Java programmers to later compare the throughput of both program versions. This will ensure that you have hard numbers that describe the results of the responsiveness of the program, as well as judge the amount of data that both algorithms processed, as the overall output is always the same.

1. The first step in creating the ideal concurrent algorithms is to analyze the parameters and the delivered performance of the sequential version of your program. Special attention must be given to sections that especially took a lot of time or used a significant portion of the available heap to the Java program. Some ideal options in this regard are the loops and decision-making elements in the program that can take a lot of time before producing the ideal results.

Other analytical elements include looking at independent parts of the program, which would not get affected, even when using concurrent programming. It may also include the object initializations and setting up the initial variables and data elements that are required for a Java program.

2. The second step in this process is to perform the concurrent designing phase. This is possible when you have analyzed your sequential code and can create a logical picture of what parts are independent and can be truly initiated with parallel program processing. There are two ways of designing concurrency. The first is to decompose the program in the form of independent tasks that can occur simultaneously. The second is the decompose the program in the form of independent data elements. You can create datasets that share resources, where you create protected access to ensure that all important data remains locked before a critical section is executed.

Always remember that the aim of your concurrent program is to create a situation where it is important to produce a benefit. If you find that adding more functional elements will create a situation where parallel execution will not be feasible, you can choose to split the program elements to make sure that the benefits are significantly available.

3. The third step to performing concurrent algorithms in Java is to carry out the implementation using the available thread, parallelism, and concurrency options. Java, with all the tools that we have mentioned in this article, is perfectly suited to provide this option using different thread classes and the ideal Java library tools.
4. The fourth step in this practice is to test your concurrency supporting code. The parallel algorithm must be tested against your sequential code, where you must compare the important results from both avenues. We will later describe how you can monitor concurrent Java applications for performing the ideal tests that deliver useful information. The testing phase will often include debugging as well, where you simply remove any programming errors.
5. This moves us to the fifth step of tuning your concurrent program. This step may not be always required, as you should only perform customized tuning if you do not get the intended benefits from your parallel processing practices. There are some important ratios that you can use when measuring the performance of concurrent applications, such as the speed up metric. This is a ratio between the execution time of the sequential program to the execution time of the concurrent program. It should always return a value greater than one, showing that there is a time benefit for using concurrent processing. The Gustafson's Law is employed when creating parallel designs that have the ability to use multiple cores for different input datasets. This is applied with the formula:

$$\text{Speedup} = N - P * (N - 1)$$

where  $P$  represents the percentage of the program code which can be parallel-processed without affecting the program integrity and  $N$  denotes the number of available cores which can all function with their separate datasets at the same time.

There are some other points that you should also keep in mind when using parallel processing with the ability to react to the real-time application needs. Remember, not all algorithms are empowered with the use of parallel processing. Programs where it is possible to run several independent threads at the same time are ideal for implementing concurrency and multiple threads for achieving the best operational performance.

There are some important points that you should always consider before using concurrent Java tools in your programs. The efficiency of your parallel program must be significantly greater than sequential processing. This is shown by either a smaller running time or the ability of the concurrent program to process more data in the same timeframe.

Your algorithm using reactive programming elements must still focus on simplicity. Remember, the final Java program should always have the simplest form, which allows for easy testing, maintenance and other programming steps, which are required before the application is ready for live use. Portability is also important where your parallel programming solution must provide the same benefits over a number of varying platforms.

The last important element is that your parallel program must have the factor of scalability. Your program should provide the same or even greater benefits when the number of available sources for the program are increased by a considerable margin. It should always be possible to scale up and create a global application from your specific code.

## 19.9.1 Ideal Tips for Concurrent Java Programs

Good concurrent programs are possible when you, as a Java developer, understand your core objective. This objective is to employ concurrency as a way to enhance the performance of your application, while ensuring that it offers significant throughput benefits. Remember, you can always learn about the code and the syntax of Java. Your learning focus should always be on understanding the benefits of employing concurrency in various Java application requirements.

Some standard practices can help ensure that you produce the optimal performance from the use of concurrency. We will describe some important concurrency tips and tricks in greater detail in the following subsections.

### 19.9.1.1 Never Assuming Thread Order

If you do not set up synchronization schemes, you can never be sure of the order of the execution of the different program threads in concurrent programming. The thread scheduler of the specific operating system decides which task is executed first if there are multiple available options, with no order specified with the use of localization.

Assuming this will never cause an issue leaves your program with conditions that give rise to data race conditions as well as thread deadlocks. There are several applications where the exact order of the tasks in the algorithm can affect the final result of the program. You cannot be sure of the reliability of your code in such a situation. With the right synchronization methods and schemes in place, you can make sure that your concurrent program always behaves in an intended manner.

### 19.9.1.2 Building Scalability Option

The main objective of your concurrent program is to take the maximum advantage of the available computer resources. This includes the available memory and the processing cores. However, computing elements may not remain the same as the supporting hardware is subject to an upgrade, especially if your client decides to scale its business project.

This means that you should build scalability in all your programs. Scalable programs are great as they provide a time proof design for your clients. Good Java programmers ensure that they never assume the number of available cores or heap sizes. The ideal program would always have a reactive programming element in it, which always finds out the available resources and then implement them in a maximum capacity in the different program functions.

This is possible with the method of `Runtime.getRuntime().availableProcessors()`, which returns the number of available processors. You can then set up your program to make use of this information in a reactive manner. This practice may increase your program overhead slightly, but it offers amazing scalability advantages, where your program can enjoy improved use of the available resources.

Designing scalability can be difficult if you perform concurrency with the use of task decomposition, rather than setting data decomposition schemes. The independent tasks can then mean that the overhead will be great because of the required

synchronization methods. The overall application performance will actually go down if your program needs to process the need for using the available processors. Always study whether you can create a dynamic algorithm which makes use of dynamic situations. Otherwise, only build limited scalability where the overhead should never exceed the advantages that can be gained from the use of additional processors.

### 19.9.1.3 Identifying Independent Code

The best performance from concurrency is only possible when you have identified all the independent tasks in your program. You should not run concurrency for tasks that depend heavily on each other, as this will simply require too much synchronization, negating the benefits of parallel processing. These tasks will run sequentially and the additional code will simply place a burden on the program. However, there are other instances where a particular task will depend on different prerequisite functions that are all independent of each other.

This situation is great for concurrency as you can run all the prerequisite execution in parallel and then place a condition for synchronization. You allow the task to initiate only when information about all the required functions has already been processed. Always remember that you cannot use concurrency in loops, as the next loop iteration often includes the use of data instances which are set up or worked on in the previous loop iteration.

### 19.9.2 High-Level Concurrency

The Java concurrency API provides various classes to perform the ideal parallelism in your programs. You can use the Lock or Thread classes, and you need to focus on the executors and fork/join facilities. This provides you access to concurrency at the highest level, like carrying out separate tasks with the use of multithreading. Here are the advantages of ensuring that concurrent tasks are specified at the highest level in your Java program:

1. The management of threads does not remain your responsibility when you allow Java to manage the intricate details of the required threads. The Flow API will manage these tasks and allow you to only create high-level functionality with a clean code.
2. Since this practice will ensure that the created threads are directly employed, they always occur in an optimized manner. The pool of threads is created according to the needs of the program once and then employed multiple times as and when required by your code. This allows you to automate several concurrency tasks.
3. The use of advanced features is possible with the use of thread groups and pools. The executors provide Future objects and ensure that you can use the main concept of concurrency for optimal benefits.
4. Your code is simple to execute in all JVM environments. The threading and other functions simply follow the available limits of the physical resources that they have available on different executing machines. This makes it easier to scale your operations and perform hardware migrations.
5. Your application is quicker to run, especially in environments that have access to the latest JVM and JRE versions. This is possible with continuous internal improvements and the ability to employ specific compiler optimizations.

#### 19.9.2.1 Use of Immutable Objects

Another ideal practice is to avoid data race conditions in your concurrent programs. This is possible by creating immutable objects in Java. These objects are special because their attributes cannot be modified after their creation. This means that they cannot get altered during multiple executions that define the concurrency practice.

The String class in Java is an excellent example of this concept. It always generates a new object whenever mathematical operators are applied in the objects of this class. Immutable objects do not need to be synchronized since it is not possible to modify the important values in these classes. The modification of any object parameters simply gives rise to a new object, which means that the main attributes always remain fixed.

Another advantage of these objects is that you can always count on the consistency of the data that you have in your program, since class defining characteristics are always secure during program executions. However, if your program ends up creating too many objects, its performance can seriously go down because it will use greater memory and reduce the throughput. Complex objects that often contain other objects defined within them can create serious issues, and you must use this practice with care.

### 19.9.3 Controlling Sequential Executions

Sequential executions will always occur in a program. Your focus should always remain on reducing the amount of time you need to keep the other tasks locked up so the critical section of the program could perform its execution. Special attention is needed to ensure that you create locks and blocks that are time sensitive and ensure that your program does not suffer from an imbalance of serial and parallel tasks during execution.

This is a situation which is often simplified with the creation of high-level concurrency. This will allow you to create shorter critical sections and avoid situations where your library can quickly supply the required code during executions. A good example would be to use the available library documentation and make the best of the Java classes. One excellent code for this is the `compute()` method which is present in the `ConcurrentHashMap` class in Java.

You should also avoid placing blocking operations in your critical section. These are operations that can block the tasks that call them for use. These tasks are then only available after a particular instance is achieved, like reading a file or outputting data to the console. The performance of your program degrades because blocking operations are slow and may stop the rest of your program from operating until the results of these operations are produced.

### 19.9.4 Avoiding Lazy initialization

One trick for successful Java programming is to employ lazy initialization scheme. It is a method where you do not create an object before it is required for the first time in your program. This practice offers the advantage of reducing the load on the available heap for the Java program. You get on in your code, only creating objects that will be required in a specific execution.

However, the same situation can be a problem in concurrent programs, where you cannot control where an object may be required for the first time, especially with aggressive multithreading strategy. If you are using singleton classes, then it will not be possible to use singular objects. You can learn more about using the on-demand holder in Java by visiting the official guide on Oracle at <https://community.oracle.com/docs/DOC-918906>, which takes care of this problem and ensures that you can take advantage of lazy initialization as much as possible.

## Summary

---

This chapter explains multiple themes that are related to the use of multithreading in the Java environment. We find that the use of multithreading brings in the application of parallel processing, which is often described in the form of concurrency in Java programming. There are several ways of implementing concurrency and designing code improvements that ensure that your Java program performs better than a typical sequential program.

We find that it is always the ideal practice to implement concurrency schemes at the highest level in your programs. In fact, with the availability of the worker thread pools and the functions offered by the executors, it is always better to leave the choice of selecting the threads and their concurrent behavior to the JVM itself. This will ensure that the ideal resources are employed each time the program operates, in various hardware environments.

We also share several important tips that help you avoid the common concurrency mistakes when programming in Java. We believe that if you focus on creating programs that use threads to match the available processors on the executing machines, it is possible to perform useful reactive programming. You can ensure that your program reads the available hardware resources before starting its working functions, and then make sure that it uses a pool of worker threads accordingly for maximum concurrency benefits.

There are times where you may never have to make difficult choices, especially with the ideal thread functionality offered by the Java Flow API. It certainly empowers concurrent programming and allows developers to focus on developing algorithms, which offer solutions to the client problems. The enhanced Java tools are sure to take care of the required thread, concurrency and reactive programming needs in each case in an ideal manner. Java is certainly an excellent choice for carrying out reactive programming, especially in various client/server applications.

In this chapter, we have learned the following concepts:

1. Reactive programming.
2. Multithreading and programming with multithreading.
3. Concurrency and advantages of concurrency
4. Concurrency API improvements.
5. Dealing with individual threads.

6. Synchronization blocks.
7. Understanding and resolving deadlocks.
8. Concurrent data structures.
9. Multithreading examples.
10. Designing concurrent Java programs.
11. Ideal tips for concurrent Java 11 programs.
12. High-level concurrency.
13. Controlling sequential executions.
14. Avoiding Lazy initialization.

In Chapter 20, we will explore the world of Spring and Hibernate. The chapter focuses on Spring Framework and Spring MVC. We will learn concepts like Inversion of Control and Dependency Injection. We will explore bean scopes like singleton and prototype. In addition, we will look into the role of controller and DispatcherServlet.

## Multiple-Choice Questions

---

1. Which of the following is a type of multitasking?
  - (a) Thread based
  - (b) Process based
  - (c) Process and thread based
  - (d) None of the above
2. Which of the following is a thread priority in Java?
  - (a) Float
  - (b) Integer
  - (c) Double
  - (d) Long
3. Which of the following methods is utilized for launching a new thread?
  - (a) `run()`
4. Which of the following is the default priority of a thread?
  - (a) 1
  - (b) 5
  - (c) 0
  - (d) 10
5. You can use the same thread object to launch multiple threads.
  - (a) True
  - (b) False

## Review Questions

---

1. What is multithreading?
2. How is multithreading useful?
3. How will you write a concurrent program?
4. What is thread life cycle?
5. How can you make a thread wait for other threads?
6. What is deadlock? How do we avoid it?
7. What is reactive programming?
8. What are concurrent data structures?
9. What are the improved concurrent APIs?
10. What are the advantages of concurrent programs?
11. How will you set thread priority to give preference to one thread over other?

## Exercises

---

1. Write a program that accepts stock stream data about current stocks and their prices. Write multithreading code to manage the stream and process in different threads.
2. Write a program for an ATM machine that dispenses money. Make sure this program uses multithreading reactive programming concepts. Also note that someone may transfer money from online backing or from a branch at the same time a user is withdrawing cash from the ATM, so you have to make sure the cash withdrawal process is secure in a way that a user should not be able to withdraw more than he/she has in his/her account.
3. Write a program using reactive programming for an online gambling application. Ensure that multiple people cannot bet on an event. You have to make sure everything happens in a timely manner.

## Project Idea

---

Create an airline ticket booking application. Any person should be able to book a flight ticket from an airline's website, ticketing counter, or through a travel agent. Please note that flight prices are not fixed, they fluctuate based on demand. Hence, price may change between checking the price and

booking a ticket. Also, tickets are limited per flight; thus, in a limited ticket availability case, the ticket may be booked by an agent while a user is trying to book it from the website. Hence, you have to make sure you use proper locks in case multiple threads are trying to book a ticket.

## Recommended Readings

---

1. Javier Fernandez Gonzalez. 2017. *Mastering Concurrency Programming with Java 9*. Packt: Birmingham
2. Mayur Ramgir and Nick Samoylov. 2017. *Java 9 High Performance*. Packt: Birmingham
3. Paul Bakker and Sander Mak. 2017. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications*. O'Reilly Media: Massachusetts

# Introduction to Spring and Spring MVC

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Spring Framework.
- Inversion of Control and Dependency Injection.
- Dependency injection variants.
- Bean Scopes such as Singleton and Prototype.
- Spring MVC.
- Role of controller and how MVC works.
- Role of Dispatcher Servlet.
- Concept of Dependency Injection and Inversion of Control.
- Maven and how to use it.
- How to connect database to perform CRUD operations.
- How to design a view to display data processed by Model through Controller.

## 20.1 | Spring Framework



In the concept of web application at a higher level, we have a presentation layer, a business layer, and a database layer. The days before Spring was invented, most of the applications were developed using the concept called *procedural programming* in which a specific task is done by calling the required libraries by the logic code. In other words, a task is achieved by performing a series of computational steps in a specific order. Spring is built on concepts called *inversion of control* and *dependency injection* (IoC and DI in short, respectively). It allows building applications from Plain Old Java Objects (POJOs) and apply other required services in a non-invasive manner to achieve the desired functionality.

### QUICK CHALLENGE

Mention all the modules of the Spring framework.

Let us explore these topics in detail to understand the inner working of Spring.

### 20.1.1 Inversion of Control

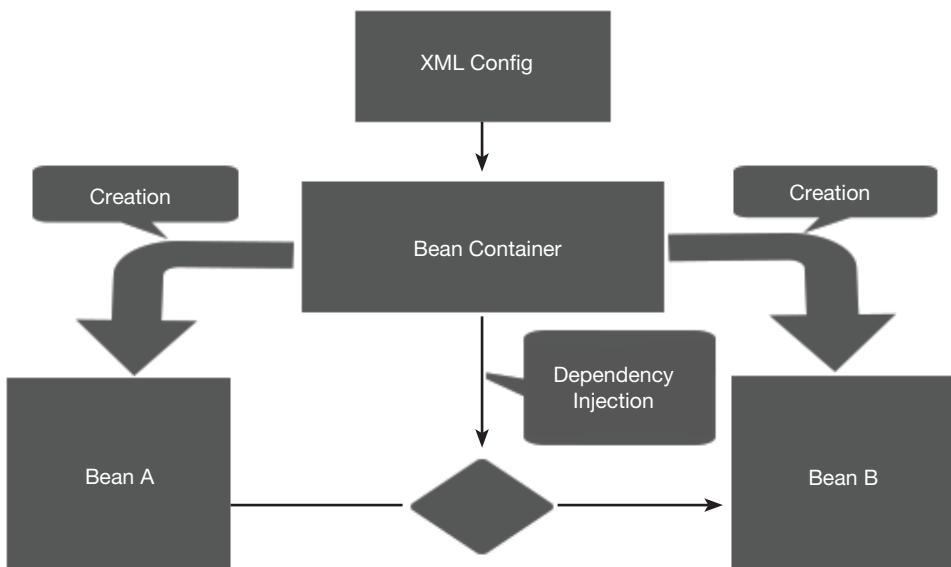
As we have seen in the traditional programming model, the logic code makes calls to the required libraries to perform a series of tasks. IoC does the exact opposite, where the control remains with the framework which executes the logic code. This promotes loose coupling, which means the objects of the functional classes do not depend on the other required objects' concrete implementation.

IoC provides various benefits such as modularity, loose coupling, etc., where the functionalities of the program get divided into various areas. This allows loose coupling where the components are not dependent on each other's implementation. Hence, it offers flexibility to modify the implementation of these objects without affecting the other parts of the application. This also enables testing teams to test the functionalities in isolation and later test the dependable functionality by mocking the object dependencies.

IoC can be achieved with the help of DI, which we will explore now.

## 20.1.2 Dependency Injection

In a typical application, there is a need to work with a lot of individual objects in order to develop a required functionality. For example, for a school registration system, we will need a student object, teacher object, classroom object, subject object, etc. These objects should be accessible in a business logic class such as Registration.java, which will enable the user to register for courses. In traditional development, you will need to initialize all these objects in order to use them. In other words, you will create these objects by yourself and need to manage the life cycle of each object for the optimum use of resources such as memory. This is where DI comes handy. DI is a structural design pattern that eliminates the need for us to initialize these required objects and manage the life cycle by ourselves. It solves this problem by injecting the required object to the constructor or setter by passing as a parameter. Figure 20.1 shows the use of DI. This DI functionality is implemented in a library called *inversion of control* (IoC) container. This IoC container is responsible for injecting the dependent objects when the bean creation is taking place. Since this operation is inverse of the traditional approach, it is called inversion of control.



**Figure 20.1** Representation of use of dependency injection.

Let us understand this better with an example. We will first take a look at the traditional approach, in which we will create an object dependency.

```

package java11.fundamentals.chapter20;
public class ObjectDependencyTraditional {
    private Product product;
    public ObjectDependencyTraditional() {
        product = new Product("My Awesome Product");
    }
}
class Product{
    private String name;

    public Product(String name) {
        this.name = name;
    }
}
    
```

In the above example, we have to instantiate the dependent object “product” within the ObjectDependencyTraditional class. Now, let us take a look at the DI approach.

```
package java11.fundamentals.chapter20;
public class ObjectDependencyWithDI {
    private ProductDI product;
    public ObjectDependencyWithDI(ProductDI product) {
        this.product = product;
    }
}
class ProductDI{
    private String name;

    public ProductDI(String name) {
        this.name = name;
    }
}
```

In the above example, IoC will inject the object of ProductDI while creating ObjectDependencyWithDI bean.

There are multiple ways you could use DI. Before exploring these, let us first understand how the IoC container works. IoC is mainly responsible for instantiating, configuring, assembling objects, and managing their life cycles. These objects are known as *beans*. Spring provides ApplicationContext interface to implement DI mechanism. This interface is a subinterface of BeanFactory interface which provides advanced configuration method to manage all types of objects. We can consider BeanFactory as the central registry which holds the configuration of all the application components. ApplicationContext extends it to add more enterprise-specific functionality. ApplicationContext interface has several implementations for various purposes; for standalone application it provides FileSystemXmlApplicationContext, for web applications it has WebApplicationContext and ClassPathXmlApplicationContext to provide a configuration for the context definition file.

Let us see how to instantiate a container to use to get the managed objects instances.

```
ApplicationContext appContext = new ClassPathXmlApplicationContext("MyApplicationContext.xml");
```

In the above example, we are using ClassPathXmlApplicationContext to get objects configurations from an XML file that is located on the class path, which is then used by the container to assemble beans at runtime.

Before we deep dive into variants of DIs, let us first explore the concept called *bean scope*. This is an important concept as it will help improve the application performance.

### Flash



How do we perform dependency injection in Spring framework?

#### 20.1.2.1 Bean Scope

In the above section, we have seen how IoC container using DI creates beans that are needed by the application. Although the IoC container creates these required beans for us, Spring provides a mechanism to have more granular control on the creation process by specifying the bean scope. The Spring framework provides five scopes. These are singleton, prototype, request, session, and global-session. Let us explore these scopes in detail:

1. **singleton:** The framework first looks for a cached instance of the bean and if it cannot find one, it creates a new one.
2. **prototype:** The framework will create a new bean instance on every request. In order to use the other three scopes listed below, you need to use a web-aware ApplicationContent.
3. **request:** This is similar to the prototype scope but only for web applications, where a new instance is created on every HTTP request.
4. **session:** As the name suggests, this scope is limited to every HTTP session. A new instance is available per session.
5. **global-session:** This scope is useful for portlet applications where a new instance is created for every global session.

Here is how you define the scope:

```
<bean id="ConstructorBasedDependencyInjectionSimpleTypeExample" class="java11.fundamentals.chapter20.ConstructorBasedDependencyInjectionSimpleTypeExample" scope="singleton" >
    <constructor-arg type="java.lang.Boolean" value="true"/>
    <constructor-arg type="java.lang.String" value="Mayur Ramgir"/>
</bean>
```

In the above example, `<bean>` definition provides an attribute called `scope` where we can specify the type of scope we need. If no “scope” attribute is provided, Spring considers the bean as “singleton”. In other words, singleton is the default bean scope.

There are multiple ways we could use DI in our application. In subsection 20.1.2.3, we will explore these ways in detail. However, before deep diving into this concept, we must first understand the concept called Autowiring. This is a mechanism used by Spring container to inject dependencies to the class.

### 20.1.2.2 Autowiring Dependencies

Spring container uses autowiring to automatically resolve dependencies. There are four modes by which we can autowire a bean in XML configuration.

1. **Default mode:** In this mode, no autowiring is used and we have to manually define the dependencies.
2. **byName mode:** In this mode, the autowiring is done using the name of the property. This name is used by Spring to look for a bean to inject.
3. **byType mode:** In this mode, the type of the property is used by Spring to look for a bean. If more than one beans are found, Spring throws an exception.
4. **Constructor mode:** In this mode, constructor of the class is used to autowire the dependencies. Spring will look for beans defined in the constructor arguments.



Are singleton beans threadsafe?

### 20.1.2.3 Variants of Dependency Injection

There are mainly three types of variants of DI, which can be used to set the dependencies for the class. These are constructor-based, setter-based, and field-based DI. Constructor-based DI is the most recommended way of injecting the required dependencies. It has many advantages over the other two methods. Setter-based injection is the next recommended way. It is highly discouraged to use field-based DI due to several disadvantages. Let us explore these in more detail to understand the reasons for these recommendations.

#### 20.1.2.3.1 Constructor-Based Dependency Injection

This method is useful to inject the required dependencies via constructor arguments. Let us see the following example that shows a class in which dependencies are supplied via constructor arguments. Please note that this is a simple POJO class which does not have any dependencies on any of Spring container’s interfaces, classes, or annotations.

```
package java11.fundamentals.chapter20;
public class ConstructorBasedDependencyInjectionExample {
    private Tyre tyre;
    private HeadLights headlight;
    public ConstructorBasedDependencyInjectionExample(Tyre tyre, HeadLights headlight) {
        this.tyre = tyre;
        this.headlight = headlight;
    }
}
class Tyre {
    public Tyre() { }
}
class HeadLights {
    public HeadLights() {}
}
```

**Constructor argument resolution:** As we have seen in the example in Section 20.1.2.3.1, we are passing two parameters to the constructor `ConstructorBasedDependencyInjectionExample`. Since these two parameters are of different types and not related by inheritance, there is no ambiguity and hence it is easier to resolve the matching process. In this case, the order in which the parameters are supplied will be used to instantiate the bean. Hence, the following configuration will work without a need to explicitly specify indexes and/or types in the `<constructor-arg>`. Let us see the following example:

```
<beans>
<bean id="ConstructorBasedDependencyInjectionExample" class="javall.fundamentals.chapter20.ConstructorBasedDependencyInjectionExample">
    <constructor-arg ref="tyre"/>
    <constructor-arg ref="headlight"/>
</bean>
<bean id="tyre" class="javall.fundamentals.chapter20.Tyre"/>
<bean id="headlight" class="javall.fundamentals.chapter20.HeadLights"/>
</beans>
```

**Constructor argument type matching:** In the above case, we are using beans as parameters, hence the type is known. This helps in matching the parameters easily. But what if we use simple types like `<value>false</value>`? It is not possible for Spring to figure out by itself whether it is a Boolean or String. Let us see the following case:

```
package javall.fundamentals.chapter20;
public class ConstructorBasedDependencyInjectionSimpleTypeExample {
    private Boolean healthy;
    private String name;
    public ConstructorBasedDependencyInjectionSimpleTypeExample(String name, Boolean healthy) {
        this.name = name;
        this.healthy = healthy;
    }
}
```

In the above case, we need to explicitly mention the type in the configuration in order for Spring to resolve the arguments.

```
<beans>
<bean id="ConstructorBasedDependencyInjectionSimpleTypeExample" class="javall.fundamentals.chapter20.ConstructorBasedDependencyInjectionSimpleTypeExample">
    <constructor-arg type="java.lang.Boolean" value="true"/>
    <constructor-arg type="java.lang.String" value="Mayur Ramgir"/>
</bean>
</beans>
```

**Constructor argument index:** In case there are multiple arguments with same type for example two `int` type arguments, we can use index attribute to specify the value for the desired argument. This will resolve the ambiguity in specifying values to the arguments.

```
<beans>
<bean id="ConstructorBasedDependencyInjectionSimpleTypeExample" class="javall.fundamentals.chapter20.ConstructorBasedDependencyInjectionSimpleTypeExample">
    <constructor-arg index="0" value="true"/>
    <constructor-arg index="1" value="Mayur Ramgir"/>
</bean>
</beans>
```

**Constructor argument name:** You may also use name attribute to specify value to the required argument. See the following example:

```
<beans>
<bean id="ConstructorBasedDependencyInjectionSimpleTypeExample" class="javall.fundamentals.chapter20.ConstructorBasedDependencyInjectionSimpleTypeExample">
    <constructor-arg name="healthy" value="true"/>
    <constructor-arg name="name" value="Mayur Ramgir"/>
</bean>
</beans>
```

### 20.1.2.3.2 Setter-Based Dependency Injection

In setter-based DI, Spring container instantiates the bean by invoking a no-argument constructor or no-argument static factory method. Once the bean is instantiated, the container then uses setter methods to inject the dependencies.

Let us see the following example to understand this better.

```
package javall.fundamentals.chapter20;
public class SetterBasedDependencyInjectionExample {
    private Camera camera;
    public SetterBasedDependencyInjectionExample() {}
    public Camera getCamera() {
        return camera;
    }
    public void setCamera(Camera camera) {
        this.camera = camera;
    }
}
package javall.fundamentals.chapter20;
public class Camera {
    public Camera() {
    }
}
```

Following is the XML configuration specify the Camera dependency:

```
<bean id = "setterBasedDependencyInjectionExample" class = "javall.fundamentals.chapter20.SetterBasedDependencyInjectionExample">
    <property name = "camera" ref = "camera"/>
</bean>
<bean id = "camera" class = "javall.fundamentals.chapter20.Camera"></bean>
```

As you can see, we need to use <property> to set the dependent bean reference. Please note that the “name” attribute must match with the field name and “ref” attribute must match the “id” attribute of the dependent bean reference.



Differentiate between setter injection and constructor injection.

### 20.1.2.3 Field-Based Dependency Injection

In field-based DI, the Spring container injects the dependencies on fields which are annotated as `@Autowired` once the class is instantiated. See the following example:

```
package java11.fundamentals.chapter20;
public class FieldBasedDependencyInjectionExample {

    @Autowired
    private Camera camera;
    public FieldBasedDependencyInjectionExample() { }
}
```

In the above example, Spring will inject the `Camera` bean since it is annotated with `@Autowired`.

This looks the simplest of all, but it is highly discouraged to use because of several drawbacks, which are as follows:

1. It is expensive to use field-based DI in place of constructor or setter as Spring uses reflection to inject the annotated dependencies.
2. It does not support final or immutable declared fields, as they will get instantiated at the time of class instantiation. Only constructor-based DI supports immutable dependencies.
3. Possibility of violating the single responsibility principle as in field-based DI for a class. It is not difficult at all to end up having a lot of dependencies without even realizing that the class now focuses on multiple functionalities and violates its single responsibility principle. This problem is clearly visible in case of constructor-based injection, as the large number of parameters in the constructor will be clearly visible and give the signal of violation.
4. It creates tightly coupled code, as the main reason of using this injection is to avoid defining getters and setters and/or constructor. This leads to a scenario where only Spring container can set these fields via reflection. This is because there are no getters or setters, so the dependencies will not be set outside the Spring container, making it dependent on using the Spring container to use reflection to set the dependencies on autowired fields.
5. It creates hidden dependencies, as these fields do not have public interfaces like getters and setters, nor are they exposed via constructors. Hence, the outside world is unaware of these dependencies.

### 20.1.2.4 Lazy Initialized Beans

This is an interesting functionality that allows initializing a bean on first request and not on the application startup. It helps in increasing application performance, as many times, you may not need all the beans loaded on the startup. This way it enhances initialization time. However, there is a danger of discovering configuration errors later on bean request. This may interrupt the running application.

Following is an example of specifying lazy initialization:

```
<bean id = "camera" class = "java11.fundamentals.chapter20.Camera" lazy-init = "true">
</bean>
```

Now that you know both IoC and DI concepts, you can see how easy it is to use the DI approach to feed the required dependencies to the bean. It greatly simplifies the application development and helps in resource management, which is crucial for performance driven applications. There is one more concept that you should know before we discuss Spring MVC. This concept is called aspect-oriented programming.

## 20.1.3 Aspect-Oriented Programming

Modularity is one of the important design principals to accomplish scalability and reusability. This can be achieved by subdividing the system into smaller parts keeping business logic separate from each other and focus on one particular functional area.

In the aspect-oriented programming (AOP) world, these standalone distinct parts are called *concerns*. Some of the functions which are distinct from the application's business logic like security, auditing, logging, caching, transactions, etc., may cross multiple areas of an application and are termed as *cross-cutting concerns*. AOP greatly helps along with DI, as DI focuses on decoupling objects from each other and AOP helps in decoupling cross-cutting concerns from the objects. We can also

relate AOP to *triggers* in programming languages like Java, .Net, Perl, etc. Similar to the trigger concept, Spring AOP offers *interceptors* which intercepts an application by inserting additional functionality on method execution such as before the method or after the method, without adding the required logic in the method itself.

**QUICK  
CHALLENGE**

Explain the different AOP concepts and terminologies.

Following are some of the important concepts we should know about AOP.

1. **Aspect:** The standalone distinct modules are called *aspect*. For example, logging aspect, auditing aspect, etc.
2. **Join point:** As the name suggests, this is the point where aspects can be injected to perform a specific task.
3. **Advice:** It is the actual code block, which will be executed before or after the method execution.
4. **Pointcut:** It is a set of one or more join points where the actual code will get executed. It is a predicate which matches join points to execute advice. Pointcuts are specified using expressions or patterns. See the following example:

```
@Aspect
public class LoggingAspectPointcutExample {
    @PointCut("execution(* java11.fundamentals.*.*(..))")
    private void logData() {}
}
```

In the above example, we have used a few annotations. Let us understand their meanings.

1. **@Aspect:** This annotation specifies that the class contains advice methods.
2. **@PointCut:** This annotation specifies the JoinPoint where an advice should be executed.
3. **execution(\* java11.fundamentals.\*.\*(..)):** This specifies on which methods the given advice should be applied.
4. **Introduction:** This enables the inclusion of new methods, fields or attributes to the existing classes. It is also known as inter-type declarations in *AspectJ*. In other words, with introduction, an aspect can assert advised objects to implement a specific interface. It can also provide implementation of this interface.
5. **Target object:** It specifies the object that is being advised by one or more aspects. This object will always be a *proxied* object as Spring AOP is implemented using runtime proxies.
6. **AOP proxy:** It denotes the object to implement the aspect contracts.
7. **Weaving:** It is a process to create an advised object by linking one or more aspects with other objects or application types. There are various ways it can be done such as compile, load, or at runtime. Spring AOP performs weaving at runtime.

### Types of Advice

There are total of five advices that can be used in the applications. Each advice has a specific purpose to accomplish. Following are descriptions of these advices:

1. **before:** This advice type makes sure to run the stated advice before the method execution on which it is specified.
2. **after:** This advice type makes sure to run the stated advice after the method execution on which it is specified irrespective of this method's outcome.
3. **after-returning:** This advice type makes sure to run the stated advice only after the successful completion of the method on which it is specified.
4. **after-throwing:** This advice type makes sure to run the stated advice only on method exits by throwing an exception on which it is specified.
5. **around:** This advice type makes sure to run the stated advice before and after the method execution on which it is specified. This type is more useful on the auditing and logging type of aspects.

**QUICK  
CHALLENGE**

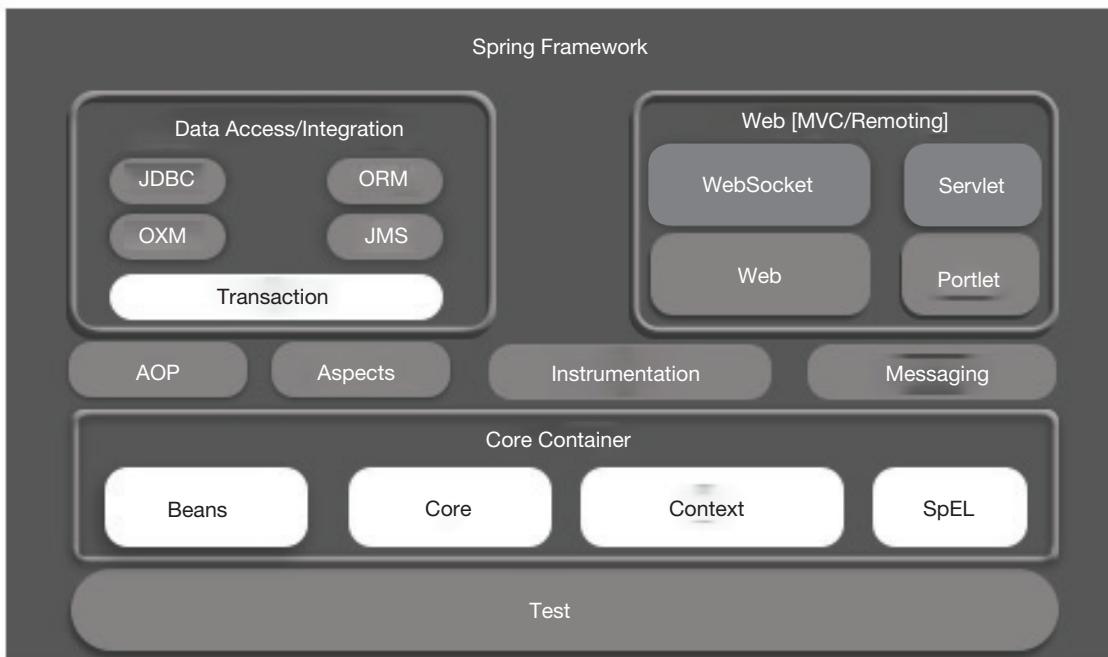
Explain Spring Aspect Oriented Programming (AOP) capabilities and goals.



## 20.2 | Spring Architecture

Now you have learned about IoC and DI, you must have comprehended that the Spring is built as a modular framework. Hence, you have an option to pick and choose the modules you need as per your requirement. Let us explore the modules offered by Spring Framework.

Overall, there are 20 modules that provide various functionalities. See Figure 20.2, which shows the Spring Framework's architecture diagram, to understand this better.



**Figure 20.2** Spring framework architecture diagram.

### 20.2.1 Core Container

As you can see, the core container acts as the base of Spring Framework, which contains modules like Beans, Core, Context, and Expression Language (SpEL). All other modules rest on top of the core container. Let us study these modules in detail.

- Core module:** This is responsible for providing fundamental features like *IoC* and *DI*, which are the crucial parts of the framework that provide modularity.
- Bean module:** This provides implementation of a factory pattern known as *BeanFactory*.
- Context module:** This module is the provider of a well-known interface called *ApplicationContext*. The Context module uses the base offered by Core and Bean modules. *ApplicationContext* is the interface that provides configuration information to the application, which is necessary to instantiates the required beans. Spring builds up this interface upon starting the application and it is read-only at run-time but it can be reloaded if required. Many classes implement this interface to provide various configuration options. *ApplicationContext* offers many bean factory methods, which can be used to access different application components. It provides various functionalities such as allowing publishing events to register listeners, loading file resources, supporting internationalization by resolving messages, etc.
- SpEL module:** This module provides a prevailing tool to query and manipulate an object graph at runtime. It supports various functionalities such as bean references, calling constructors, method invocation, collection projection, collection selection, relational operators, regular expressions, Boolean and relational operators, assignment, accessing properties, array construction, accessing arrays, accessing lists, accessing maps, literal expression, class expressions, user defined functions, inline maps, inline lists, variables, templated expressions, and ternary operator. It also offers various interfaces and classes such as ExpressionParser interface, EvaluationContext interface, Expression interface, SpelExpression class, SpelExpressionParser class, and StandardEvaluationContext.



Does Spring provide transaction management?

## 20.2.2 Data Access/Integration

This section looks after accessing, storing, and manipulating data. This layer contains modules such as ORM, OXM, JDBC, JMS, and Transaction. Following are the descriptions of each module.

1. **ORM module:** ORM stands for object-relationship mapping, which offers an easy way to persist objects without writing database queries. It binds objects to the database tables, which hides the details of SQL queries from application logic. With ORM, objects can deal with other objects without worrying about underlying persistence mechanism. Spring does not offer ORM implementation but it provides a mechanism to integrate with external ORM tools like Hibernate, JPA, JDO, iBatis, etc. In the Chapter 21, we will explore Hibernate in detail.
2. **JDBC module:** This module provides an abstraction layer for JDBC, which is a Java API to interact with database to perform CRUD operations (CRUD stands for create, read, update, and delete). It uses JDBC drivers to connect with different providers databases.
3. **OXM module:** OXM stands for Object XML Mapping, which offers a mechanism to convert an object to an XML document and vice versa. XML is one of the important formats of transferring data to remote applications. With OXM, it is easier to convert the transmitted data into an object and prepare objects to transfer over network. The conversion from object to XML known as XML Marshalling or XML Serialization, and from XML to object known as Unmarshalling or Deserialization. This module is an extendible one and can integrate with other frameworks such as Castor, JAXB, and XStream.
4. **JMS module:** In distributed and modular systems, messaging plays a crucial role for communication between loosely coupled elements. Java provides an API called Java Message Service (JMS), which offers a mechanism to create, send, and read messages in a reliable and asynchronous way.
5. **Transaction management module:** Transaction management is a critical feature in RDBMS to promise data integrity and consistency. Database transactions are a series of actions that are considered as a single unit of work, which must complete totally or not at all. This module offers a reliable abstraction for transaction management which not only supports declarative transaction management but also offers a programmatic transaction management simpler than Java Transaction API (JTA). This module is consistent across different APIs such as JTA, Hibernate, Java Persistent API (JPA), JDBC, and Java Data Objects (JDO).

### QUICK CHALLENGE

Explain the benefits of ORM.

## 20.2.3 Web Container

This section provides various frameworks that are useful for web related applications, which offers integration with other popular MVC frameworks such as Struts and JSF. It has various modules such as Web, Web-Servlet, Web-Sockets, and Web-Portlet. Let us explore in detail.

1. **Web module:** This module offers web-related functionalities such as multiple file uploads, integration to IoC using a servlet, and web application context.
2. **Web-servlet module:** This module is an implementation of Model–View–Controller (MVC) pattern. It offers a clean separation between domain model code and web interface.
3. **Web-socket module:** This module provides support of Web-Socket, which is a bidirectional communication between server and client.
4. **Web-portlet module:** This module provides similar functionality as Web-WebSocket module for implementing MVC in a portlet environment.



What is the role of web container?

## 20.2.4 Other Modules

1. **AOP module:** This module provides Aspect Oriented Programming support to Spring Framework. As we have learned in Section 21.1.3, AOP provides a simple mechanism to implement cross-cutting concerns.
2. **Aspects module:** This module supports integration with AspectJ.
3. **Testing module:** This module provides provision to implement integration and unit tests. With the help of this module, JUnit or TestNG type of frameworks can be used in Spring Framework.
4. **Instrumentation module:** This module provides class instrument support and classloader implementations, which are useful in various application servers.
5. **Messaging module:** This module provides foundation for messaging-based applications, which contains a set of annotations for mapping messages to methods.

Due to the modularity of the Spring Framework, you are free to choose the modules you need without breaking any core framework's functionality.

## 20.3 | Spring MVC



Spring MVC is one of the most popular Java website frameworks to program website applications. Similar to other industry technologies, it makes use of the model view design. All the general Spring features such as DI and IoC are implemented by Spring MVC.

To use the Spring framework, Spring MVC makes use of class DispatcherServlet. This class processes an incoming request and assigns it with models, controller, views, or any of the right resource.

If you are unfamiliar with MVC pattern, then bear in mind that it is a software architectural pattern. The data of the application is referred to as the model which can be a single or multiple objects. For the business logic, a controller is used which acts as a supervisor. The end user is provided a perspective through a "view".

In Spring MVC, any class which uses the annotation "@Controller" is the controller of the application. For views, a combination of JSP and JSTL can be utilized, though developers can also use other technologies. For the front controller, the DispatcherServlet class is used in Spring MVC.

When a request arrives, it is discovered by the DispatcherServlet class. This class gets the required information from the XML files and delivers the request to the controller. Subsequently, the controller generates a ModelAndView object. Afterwards, the DispatcherServlet class processes the view resolver and calls the appropriate view.

### 20.3.1 DispatcherServlet

The DispatcherServlet is used for request processing. It has to be mapped and declared in accordance with the Servlet specification. This servlet utilizes configurations in Spring to get information related to the delegate components for exception handling, view resolution, and request mapping.

#### 20.3.1.1 Context Hierarchy

For configuration purposes, the DispatcherServlet requires the WebApplicationContext. It is linked with the Servlet and ServletContext. It is also attached to the ServletContext so the methods which are associated with it can utilize the RequestContextUtils.

For most of the applications, one WebApplicationContext is enough. However, sometimes there is a context hierarchy in which a single root WebApplicationContext is used for sharing between more than one instance of DispatcherServlet. Each of them has its configuration of own child WebApplicationContext.

Generally, the root WebApplicationContext stores data repositories, business services, and other infrastructure beans. These are shared between the Servlet instances. It is possible to override these beans as they are inherited via the WebApplicationContext of the Servlet children.

#### Special Bean Types

1. **HandlerMapping:** It is used with a handler for mapping requests. It uses the HandlerMapping implementation.
2. **HandlerAdapter:** It uses the DispatcherServlet for invoking handlers which are mapped with a request.

3. **HandlerException resolver:** It is used for resolving exceptions where they are mapped with handlers and error views of the HTML.
4. **View resolver:** It is used to resolve the view names which rely on logical Strings.
5. **Locale resolver:** It is used for resolving a client's locale or timezone.
6. **Themes resolver:** It is used to resolve the themes of a web application.
7. **MultipartResolver:** It is used for abstraction in order to help with parsing a request which has multiple parts.
8. **FlashMapManager:** It is used for storing and retrieving the input/output FlashMap.

### 20.3.2 Spring MVC Processing

The DispatcherServlet is used for the following:

1. It searches the WebApplicationContext and attaches the request in the form of an attribute so other components such as controller can use it.
2. The request is bound to the locale resolver which allows the process' elements in resolving the locale while processing and preparing data, rendering the view, or other requests.
3. Requests are bound to the theme resolver which allows views and other elements to use the theme.
4. When a multipart file resolver is defined, all the parts of the request are processed. After multipart are discovered, the MultipartHttpServletRequest is wrapped to help with processing.
5. When the right handler is found, controllers, preprocessors, postprocessors, and others in the handler's executive chain are executed so the model is prepared or rendered. For annotated controllers, it is possible to render the response.
6. The view is rendered when a model is returned. When it is not returned, there is no need for view rendering as the request is already processed.
7. The declared beans in HandlerExceptionResolver resolve exceptions which come up in the request processing.

The Spring DispatcherServlet helps in returning the “last-modification-date”, which is defined in the Servlet API. In order to assess this date, a simple processing is used.

1. The DispatcherServlet searches for the right handler mapping.
2. The DispatcherServlet then checks if the handler has implemented the “LastModified” interface.
3. If verified, the “long getLastModified” method value of the interface is sent to the client.



Can you configure Spring application with Configuration file and Annotations?

## 20.4 | Interception



In all of the Spring MVC HandlerMapping implementations, interceptors are supported. They are used to implement certain functionality for specific requests. For instance, they can be used to check for a principal. It is necessary for an interceptor to implement “HandlerInterceptor”, which is included in the org.springframework.web.servlet having the following methods for preprocessing and post-processing.

1. **preHandle():** It is used for the time period before the handler is executed.
2. **postHandle():** It is used when the handler has been executed.
3. **afterCompletion():** It is used after the request has been completed the execution.

## 20.5 | Chain of Resolvers



When you declare multiple beans of HandlerExceptionResolver, you can create an exception resolver chain in your SpringMVC application where you have to configure and define the order properties accordingly. If the order property is high, then the exception resolver is processed accordingly. With respect to the HandlerExceptionResolver, it returns the following.

1. An empty ModelAndView in case the resolver handled the exception.
2. A Model and View which refers to an error view.
3. In case the exception is not resolved, it returns “null” for other resolvers. On the other hand, if the exception is still staying till the end, the Servlet container can be used to bubble it up.



Can you manage concurrent sessions using Spring MVC? If so, what are the things you need to consider?

## 20.6 | View Resolution



The View and ViewResolver interfaces are used by the SpringMVC, which helps in rendering models present in a browser while avoiding dependence on a certain technology for viewing. The ViewResolver is used to map actual views and view names. The View manages the preparation data before it is passed over to a certain technology for viewing.

It is possible to use multiple resolver beans to chain the view resolvers. Sometimes, the order property is used for defining ordering with it. To configure the view resolution, you just have to add the ViewResolver beans in the configuration of your SpringMVC application.

### A Basic Example

Consider the following example for a basic Spring Web MVC project. Use Maven and add dependency in the pom.xml file.

```
<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 http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.introspring</groupId>
<artifactId>SpringMVC</artifactId>
<packaging>war</packaging>
<version>0.0.1-SNAPSHOT</version>
<name>SpringMVC Example</name>
<url>http://maven.apache.org</url>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>servlet-api</artifactId>
<version>3.0-alpha-1</version>
</dependency>
</dependencies>
<build>
<finalName>SpringMVC</finalName>
</build>
</project>
```

Now generate a class for the controller and add the following two annotations. The @Requestmapping annotation is required for mapping the appropriate URL name with the class. The @Controller class is simply used to treat a class as the controller.

```

package com.introspring;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class ContClass {
@RequestMapping("/")
    public String show()
    {
        return "index";
    }
}

```

In the XML file, make sure to write about the DispatcherServlet class which is serves as the front controller.

```

<?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/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
    <display-name>Model View Controller</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>

```

In the spring-servlet.xml file, define the components for the views. This XML file must be placed in the WEB-INF directory.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!-- Specify the functionality to scan components -->
    <context:component-scan base-package="com.introspring" />

    <!--Specify the functionality for formatting, validation, and conversion -->
    <mvc:annotation-driven/>
</beans>

```

Generate an index JSP page and write the following.

```
<html>
<body>
<p>We are learning Spring MVC</p>
</body>
</html>
```

As an output, you can see “We are learning Spring MVC” text on your page. This message was delivered to you through the logic of the Controller.

## 20.7 | Multiple View Pages

In this example, we would apply redirection between two view pages. Begin by adding dependencies in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
```

Afterward, you have to generate a page for the request. To do this, create a JSP page and add a hyperlink in it.

```
<html>
<body>
<a href="Success">Hit This Link...</a>
</body>
</html>
```

Now generate a class for the controller which can process the JSP pages and return them. For mapping of the class, use the @Requestmapping annotation. We would add our “href” value in the following code so our Controller can easily manage and view our HTML elements.

```
package com.ith;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class ContClass {
    @RequestMapping("/success")
    public String reassign()
    {
        return "view";
    }
    @RequestMapping("/success again")
    public String show()
    {
        return "final";
    }
}
```

In the web.xml file, place a controller entry.

```
<?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/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

Now, define the bean for XML file. You have to add the view resolver in the component of the view. For the ViewResolver, the InternalResourceViewResolver class can be used. The XML file must be placed in the WEB-INF directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <!--Add functionality to scan components -->
    <context:component-scan base-package="com.ith" />

    <!--Add functionality for formatting, validation, and conversion -->
    <mvc:annotation-driven/>
    <!--Specify the view resolver for the Spring MVC -->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceView-
    wResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
</beans>
```

Now generate the components for the view. In the view.jsp, write the following.

```
<html>
<body>
<a href="successagain">Spring Tutorial</a>
</body>
</html>
```

In the last.jsp, write the following.

```
<html>
<body>
<p>Hello User, Let's Learn Spring MVC to Master the Java Backend</p>
</body>
</html>
```

When you run this application, you would first get a web page with a hyperlink. When you will click on it, it sends a request which is managed by the controller to reply with a response in the form of another page. Hit the link on the new page and you would be again redirected to another page, which would show you a welcome message. In this way, we have successfully gone over multiple web pages in Spring MVC.

## 20.8 | Multiple Controllers

Till now, we have been using a single controller in our examples. However, it is possible to generate multiple controllers in Spring MVC at the same time. Each of the controller class must be mapped properly with the @Controller annotation.

Begin by adding the dependencies in the pom.xml file as we have done before.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
```

Then generate your initial page which would be initially viewed by the user. We would add two different hyperlinks for each of our controllers.

```
<html>
<body>
<a href="hiuser1">First User</a> ||
<a href="hiuser2">Second User</a>
</body>
</html>
```

Generate two classes for controller which can process and return the specified view page. For the first class, write the following code.

```
package com.ith;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class User1 {
    @RequestMapping("/hiuser1")
    public String show()
    {
        return "vp1";
    }
}
```

For the second class, write a similar piece of code.

```
package com.ith;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class User2 {
    @RequestMapping("/hiuser2")
    public String show()
    {
        return "vp2";
    }
}
```

Place an entry for the controller in the web.xml file.

```
<?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">
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

Then specify the bean in the XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- Add functionality to scan component-->
    <context:component-scan base-package="com.ith" />
    <!-- Add functionality for validation, formatting, and conversion -->
    <mvc:annotation-driven/>
    <!--Specify the view resolver in Spring MVC -->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
</beans>
```

Generate the components for the view. In the first vp1.jsp, write the following.

```
<html>
<body>
<p> In this example we have used two controllers.</p>
</body>
</html>
```

In the second vp2.jsp page, write the following code.

```
<html>
<body>
<p> In this example we have used two view pages.</p>
</body>
</html>
```

Run this code. In the output, the user would be presented two links. When the user clicks on the first page, they would be redirected to a new page and displayed a text. Clicking on the second link will take the user to a different page with different text. This is exactly how it happens in real world browsing. Hence, whenever you are clicking on links on the websites, then behind the scenes, each link is configured to redirect to a different page by their respective controllers.



Can you add security on Spring controllers to avoid denial of service attack? Explain.



## 20.9 | Model Interface

In the Spring MVC ecosystem, the model serves as a container which can be used to store the application's data. This data can be anything such as a record from the DB, an object, or simply a String. The controller must have the Model interface. The HttpServletRequest object processes the data given by a user and sends it to this interface. Any view page can then access that piece of data from model's interface.

As an example, let us generate a login page which has a username and password, a common use case in the web world. For validation, we can define a certain value. Add the required dependencies in the pom.xml.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
```

Now generate the login page which would get the user's credentials. We will name it as index.jsp.

```

<html>
<body>
<form action="hiuser">
UserName : <input type="text" name="name"/> <br><br>
Password : <input type="text" name="pass"/> <br><br>
<input type="submit" name="submit">
</form>
</body>
</html>

```

Now, generate the controller class in which the HttpServletRequest would be responsible to intercept the data which is typed by the user on the HTML form. The Model interface saves the data for the request and sends it to the view page.

```

import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
public class ContClass{
    @RequestMapping("/hiuser")
    public String show(HttpServletRequest r,Model md)
    {
        //read the data from the form
        String name=r.getParameter("name");
        String pass=r.getParameter("pass");
        if(pass.equals("abc12"))
        {
            String message="Welcome "+ name;
            //add a message to the model
            md.addAttribute("message", message);
            return "vp";
        }
        else
        {
            String msg="We apologize Mr. "+ name+". This is the wrong password";
            md.addAttribute("message", message);
            return "ep";
        }
    }
}

```

Now, open the web.xml file and add the controller's entry.

```

<?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/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>

```

Afterward, configure the bean by setting up the spring-servlet.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- Add functionality to scan component-->
    <context:component-scan base-package="com.ith" />
    <!-- Add functionality for validation, formatting, and conversion -->
    <mvc:annotation-driven/>
    <!--Specify the view resolver in Spring MVC -->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
</beans>
```

For the components of the view, add two .jsp pages in the WEB-INF/jsp directory. The first page vp.jsp should look like the following. See how we are using the backend code to display our code dynamically on user's view by adding the "\${message}" parameter.

```
<html>
<body>
${message}
</body>
</html>
```

The second page ep.jsp should show something like the following.

```
<html>
<body>
${message}
<br><br>
<jsp:include page="/index.jsp"></jsp:include>
</body>
</html>
```

When the user gets the username page, then there are two possible scenarios.

1. If the user types the correct password "abc12" and hits the submit button, then the controller would take them into a new page with a greeting.
2. On the other hand, a wrong password means that they would remain in the same page with a warning.

The basic idea behind the "login" functionality you see in websites is pretty much the same.

## 20.10 | RequestParam

The @RequestParam annotation reads data from the form and applies automatic binding for the method's available parameters. It does not consider the HttpServletRequest object for reading data. The annotation also applies mapping for the request parameter. If the type of the parameter in the method is "Map" along with the definition for the name of the

request parameter, then that parameter is changed into a Map. Otherwise, the name and values for the request parameter are placed in the map parameter.

As an example, let us generate a login page. Begin by placing the required dependencies in the pom.xml.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
```

Now, generate the page for the request which would take the credentials of the user. This page is saved as index.jsp.

```
<html>
<body>
<form action="hiuser">
Student Name : <input type="text" name="name"/> <br><br>
Password : <input type="text" name="pass"/> <br><br>
<input type="submit" name="submit">
</form>
</body>
</html>
```

Now, generate a controller class. In this class, the @RequestParam annotation reads the data from the form and applies binding for the request parameter.

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class ContClass {
    @RequestMapping("/hiuser")
    //read the provided form data
    public String display(@RequestParam("name") String name, @RequestParam("pass") String pass, Model md)
    {
        if(pass.equals("abc12"))
        {
            String message="Welcome "+ name;
            //write a message for the model
            md.addAttribute("message", message);
            return "vp";
        }
        else
        {
            String msg="We apologize "+ name+". The typed password is wrong";
            md.addAttribute("message", message);
            return "ep";
        }
    }
}
```

For the vp page, write the following.

```
<html>
<body>
${message}
</body>
</html>
```

For the ep page, write the following.

```
<html>
<body>
${message}
<br><br>
<jsp:include page="/index.jsp"></jsp:include>
</body>
</html>
</html>
```

The output is similar to the previous example. However, the controller's working changed. In this example, we did not use the HttpServletRequest class. Instead, we used our @RequestParam annotation for our HTML elements. Through the annotation, we were able to apply automatic binding on HTML elements.

## 20.11 | Form Tag Library

Form tags in Spring MVC are the basic components of web pages. These tags can be reused and reconfigured according to the requirements of the users. They are extremely helpful for JSP for the readability and maintainability factors.

The library for the form tags is in the spring-webmvc.jar. For turning on the library's support, some configuration is required. Place the following code in the start of the JSP web page.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

The container tag is a sort of parent tag which stores all of the library's tag. Such tag creates a form tag for HTML.

```
<form:form action="nextFormPath" modelAttribute=?abc?>
```

Following are one of the most common form tags in Spring MVC.

1. **form:form** – Used to store all of the other tags.
2. **form:input** – Creates the text field for the user input.
3. **form:radiobutton** – Creates the radio button which can be marked or unmarked by the user.
4. **form:checkbox** – Creates the checkboxes which can be checked or unchecked by the user.
5. **form:password** – Creates a field for user to type password.
6. **form:select** – Creates a drop-down list for the user to choose an option.
7. **form:textarea** – Creates a field for text which spans multiple lines.
8. **form:hidden** – Creates an input field which is hidden.

## 20.12 | Form Text Field

The form text field creates an HTML input tag through the use of the bound value. For instance, consider a form where users can type their details to make a reservation in a restaurant. This example mirrors the real world reservations and booking structure.

To begin with, add the required dependencies in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>

<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
```

Generate a bean class now. This class can store variables and the getter setter methods for the input fields of the form.

```
public class Reservation {
    private String fName;
    private String lName;
    public Reservation() { }
    public String getFname() {
        return Fname;
    }
    public void setFname(String Fname) {
        this.Fname = Fname;
    }
    public String getLname() {
        return Lname;
    }
    public void setLname(String Lname) {
        this.Lname = Lname;
    }
}
```

Now add a Controller class.

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotationModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@RequestMapping("/reservation")
@Controller
public class ContClass {
    @RequestMapping("/bkfrm")
    public String bkfrm(Model md)
    {
        //generate an object for reservation
        Reservation rsv=new Reservation();
        //pass the object to the model
        md.addAttribute("reservation", rsv);
        return "rsv-page";
    }
    @RequestMapping("/subfrm")
    // @ModelAttribute applies binding for the data of the form to the object
    public String submitForm(@ModelAttribute("reservation") Reservation rsv)
    {
        return "confirmation-form";
    }
}
```

You can then add the controller's entry in the web.xml file.

```
<?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">
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

In the spring-servlet.xml file, place the following code.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!--Enables the scanning of components -->
    <context:component-scan base-package="com.ith" />
    <!--Enables formatting, validation, and conversion -->
    <mvc:annotation-driven/>
    <!-- Define Spring MVC view resolver -->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalRe-
sourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
</beans>
```

To generate the requested page, place the following code in the index.jsp file.

```
<!DOCTYPE html>
<html>
<head>
    <title>Reservation Form for the Restaurant</title>
</head>
<body>
<a href="reservation/bkfrm">Click this link to get a booking at the restaurant</a>
</body>
</html>
```

For the reservation page, place the following code.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<!DOCTYPE html>
<html>
<head>
    <title>Restaurant Form</title>
</head>
<h3>Reservation Form for Restaurant</h3>
<body>
    <form:form action="subfrm" modelAttribute="reservation">
        First name: <form:input path="Fname" />
        <br><br>
        Last name: <form:input path="Lname" />
        <br><br>
        <input type="submit" value="Submit" />
    </form:form>
</body>
</html>
```

In the confirmation page, place the following code.

```
<!DOCTYPE html>
<html>
<body>
<p>Your reservation is confirmed successfully. Please, re-check the details.</p>
First Name : ${reservation.Fname} <br>
Last Name : ${reservation.Lname}
</body>
</html>
```

In the output, a hyperlink will take a user to a reservation form for the restaurant. When the user completes typing the details and submits it, then they are displayed their information as part of the rechecking procedure.

## 20.13 | CRUD Example

If you are learning a web framework, then it is important to equip yourself with enough knowledge that you can make a basic CRUD application. Such functionality is one of the most common client requirements and therefore getting the hang of it can be extremely useful for your career. To begin with, generate a new table in your database titled “students”. Configure and enter the following fields in it.

1. ID
2. Name
3. GPA
4. Department

Now come to Eclipse IDE and begin the management of your dependencies by adding the following in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.11</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
```

Subsequently, you have to create your bean class “Student” which contains variables according to the columns of your database.

```
public class Student {
    private int id;
    private String name;
    private float GPA;
    private String department;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public float getGPA() {
        return GPA;
    }
    public void setGPA(float GPA) {
        this.GPA = GPA;
    }
    public String getDepartment() {
        return department;
    }
    public void setDepartment(String department) {
        this.department = department;
    }
}
```

For the controller class, write the following code.

```
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import com.ith.beans.Student;
import com.ith.dao.StudentDao;
@Controller
```

```

public class StudentController {
    @Autowired
    StudentDao sDao;//Use the XML file for dao injection
    /* To take input, it generates a form. You can also see the reserved request attribute
    in the form of "command". This attribute shows the data pertaining to the object in the
    form.
    */
    @RequestMapping("/studentform")
    public String displayform(Model md){
        md.addAttribute("command", new Student());
        return "studentform";
    }
    /* This is done in order to use the database for storing data. The model
    object receives request data from the @ModelAttribute. It is also necessary to add
    RequestMethod.Post method as by default, the request type is set to GET.
    */
    @RequestMapping(value="/store",method = RequestMethod.POST)
    public String store(@ModelAttribute("student") Student student){
        sDao.store(student);
        return "redirect:/viewstudent";//
    }
    /* To show the student list which are stored in the model object */
    @RequestMapping("/viewstudent")
    public String viewstudent(Model md){
        List<Student> list=sDao.getStudents();
        md.addAttribute("list",list);
        return "viewstudent";
    }
    /* It shows the data of object from the form in accordance with the provided id.
    To add data in the variable, we have used the @PathVariable. */
    @RequestMapping(value="/editstudent/{id}")
    public String edit(@PathVariable int id, Model md){
        Student student=sDao.getStudentById(id);
        md.addAttribute("command",student);
        return "studentupdateform";
    }
    /* It edits the object in the model. */
    @RequestMapping(value="/editsave",method = RequestMethod.POST)
    public String editsave(@ModelAttribute("student") Student student){
        sDao.update(student);
        return "redirect:/viewstudent";
    }
    /* It is used for the deletion of a record. */
    @RequestMapping(value="/deletetestudent/{id}",method = RequestMethod.GET)
    public String delete(@PathVariable int id){
        sDao.delete(id);
        return "redirect:/viewstudent";
    }
}

```

Create a DAO class like the following.

```

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import com.ith.beans.Student;

public class StudentDao {
    JdbcTemplate tpe;

    public void setTemplate(JdbcTemplate tpe) {
        this.tpe = tpe;
    }
    public int save(Student s){
        String sql="insert into students (name,GPA,department) values ('"+s.
getName()+"','"+s.getGPA()+"','"+s.getDepartment()+"')";
        return tpe.update(sql);
    }
    public int update(Student s){
        String sql="update students set name='"+s.getName()+"', GPA='"+s.
getGPA()+"',department='"+s.getDepartment()+"' where id='"+s.getId()+"';
        return tpe.update(sql);
    }
    public int delete(int id){
        String sql="delete from students where id='"+id+"'";
        return tpe.update(sql);
    }
    public Student getStudentById(int id){
        String sql="select * from students where id=?";
        return tpe.queryForObject(sql, new Object[]
{id},new BeanPropertyRowMapper<Student>(Student.class));
    }
    public List<Student> getStudents(){
        return tpe.query("select * from students",new RowMapper<Student>(){
            public Student mapRow(ResultSet rs, int row) throws SQLException {
                Student s=new Student();
                s.setId(rs.getInt(1));
                s.setName(rs.getString(2));
                s.setGPA(rs.getFloat(3));
                s.setDepartment(rs.getString(4));
                return s;
            }
        });
    }
}

```

In the web.xml file, add the controller.

```

<?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">
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>

```

In the XML file, specify the bean.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc.xsd">
<context:component-scan base-package="com.ith.controllers"></context:component-scan>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="prefix" value="/WEB-INF/jsp/"></property>
<property name="suffix" value=".jsp"></property>
</bean>

<bean id="ds" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
<property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
<property name="url" value="jdbc:mysql://localhost:3306/test"></property>
<property name="username" value=""></property>
<property name="password" value=""></property>
</bean>

<bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
<property name="dataSource" ref="ds"></property>
</bean>

<bean id="sDao" class="com.ith.dao.StudentDao">
<property name="template" ref="jt"></property>
</bean>
</beans>
Generate a requested HTML page.
<a href="studentform">Add Students</a>
<a href="viewstudent">View Students</a>
In the JSP student form, add the following.
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<h1>Add New Student</h1>
<form:form method="post" action="save">
<table >
<tr>
<td> StudentName : </td>
<td><form:input path="name" /></td>
</tr>
<tr>
<td>GPA :</td>
<td><form:input path="GPA" /></td>
</tr>
<tr>
<td>Department :</td>
<td><form:input path="department" /></td>
</tr>
<tr>
<td> </td>
<td><input type="submit" value="Save" /></td>
</tr>
</table>
</form:form>

```

By changing the project name in the following file, add the following code in the studenteditform.jsp file.

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

    <h1>Edit Student</h1>
    <form:form method="POST" action="/SpringCRUDExample/editsave">
        <table >
            <tr>
                <td></td>
                <td><form:hidden path="id" /></td>
            </tr>
            <tr>
                <td>Name : </td>
                <td><form:input path="name" /></td>
            </tr>
            <tr>
                <td>GPA :</td>
                <td><form:input path="GPA" /></td>
            </tr>
            <tr>
                <td>Department :</td>
                <td><form:input path="department" /></td>
            </tr>
            <tr>
                <td> </td>
                <td><input type="submit" value="Edit Save" /></td>
            </tr>
        </table>
    </form:form>
```

In the viewstudent file, add the following code.

```
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

    <h1>Student List</h1>
    <table border="2" width="70%" cellpadding="2">
        <tr><th>Id</th><th>StudentName</th><th>GPA</th><th>Department</th><th>Edit</th><th>Delete</th></tr>
        <c:forEach var="student" items="${list}">
            <tr>
                <td>${student.id}</td>
                <td>${student.name}</td>
                <td>${student.GPA}</td>
                <td>${student.department}</td>
                <td><a href="editstudent/${student.id}">Edit</a></td>
                <td><a href="deletestudent/${student.id}">Delete</a></td>
            </tr>
        </c:forEach>
    </table>
    <br/>
    <a href="studentform">Add New Student</a>
```

When this application is run, the user gets two links; they can either add a student or see the student table. Since this is the first time you are using this application, first focus on adding the entries. Click the add button, and then you are provided with

a form to type the required details. When these details are completed, then it can be saved and the data would then be stored in the database. A user can then use this step multiple times to populate the table. After generating a table, you can modify or delete the data in the table. Click on either one of them and perform your required task. If you choose “Edit” then you can revisit the form and modify any detail. On the other hand, if you click on “Delete” then you can permanently remove the record from your database. The database would update accordingly.

You can use the logic implemented in this application in various other applications. For instance, instead of creating a student list, you can modify the details and make it a grocery shopping list where you can always adjust your items.

## 20.14 | File Upload in Spring MVC

In Spring MVC, you can easily incorporate the functionality to upload files in various formats. Consider the following example.

At the start, you have to download and load the Spring Core, Spring Web, commons-fileupload.jar, and commons-io.jar file.

Afterward, place the commons-io and fileupload.jar files. In the spring-servlet.xml file, write the following:

```
<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
Generate a form for the submission of file.
<form action="savefile" method="post" enctype="multipart/form-data">
Choose a File: <input type="file" name="file"/>
<input type="submit" value="Upload the File"/>
</form>
```

For the controller,

```
@RequestMapping(value="/savefile",method=RequestMethod.POST)
public ModelAndView upload(@RequestParam CommonsMultipartFile file,HttpSession session){
    String pth=session.getServletContext().getRealPath("/");
    String fname=file.getOriginalFilename();
    System.out.println(pth+" "+fname);
    try{
        byte b[]={};
        BufferedOutputStream bos=new BufferedOutputStream(
            new FileOutputStream(path+"/"+fname));
        bos.write(b);
        bos.flush();
        bos.close();
    }catch(Exception e){System.out.println(e);}
    return new ModelAndView("upload-success","fname",pth+"/"+fname);
}
```

To show your image in JSP, add the following code.

```
<h1>The Upload Is Successful.</h1>

To store the files in your project, generate a directory, "images".
<a href="uploadform">Upload the Image</a>
```

In the Student class, write the following:

```

import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import org.apache.commons.fileupload.disk.DiskFileItemFactory;
import org.apache.commons.fileupload.servlet.ServletFileUpload;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.multipart.commons.CommonsMultipartFile;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class StudentController {
    private static final String UPLOAD_DIRECTORY = "/images";

    @RequestMapping("uploadform")
    public ModelAndView uploadForm() {
        return new ModelAndView("uploadform");
    }

    @RequestMapping(value="savefile",method=RequestMethod.POST)
    public ModelAndView saveimage( @RequestParam CommonsMultipartFile file,
        HttpSession session) throws Exception{
        ServletContext context = session.getServletContext();
        String pth = context.getRealPath(UPLOAD_DIRECTORY);
        String fname = file.getOriginalFilename();
        System.out.println(pth+" "+fname);
        byte[] b = file.getBytes();
        BufferedOutputStream str =new BufferedOutputStream(new FileOutputStream(
            new File(pth + File.separator + fname)));
        str.write(b);
        str.flush();
        str.close();
        return new ModelAndView("uploadform","filesuccess","The file is saved successfully!");
    }
}

```

In the web.xml file, write the following code:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>

```

Subsequently, build a bean and type the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package="com.ith"></context:component-scan>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="prefix" value="/WEB-INF/jsp/"></property>
<property name="suffix" value=".jsp"></property>
</bean>

<bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

</beans>
```

The action for the form must be “post” in the following format.

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<!DOCTYPE html>
<html>
<head>
<title>Upload the Image</title>
</head>
<body>
<h1>File Upload Example in Spring MVC</h1>

<h3 style="color:blue">${filesuccess}</h3>
<form:form method="post" action="savefile" enctype="multipart/form-data">
<p><label for="image">Pick any image</label></p>
<p><input name="file" id="fileToUpload" type="file" /></p>
<p><input type="submit" value="Upload"></p>
</form:form>
</body>
</html>
```

In the output, the user first got a hyperlink. After clicking and redirecting into a new web page, the user was able to find a screen with a file upload option. When you upload a file or an image through this button, it would not show in the browser (or the client side). Instead, the file or image is sent to the server. Have your path printed via the console of the server and check the file.

## 20.15 | Validation in Spring MVC

Validation in Spring MVC is required for the restriction of user input so only the authorized users can proceed through a website.

In order to execute this validation, the Bean Validation API is used by the Spring MVC. This validation can be used for both the client-side and server-side programming. To use annotations and implement restrictions on object model, we use the Bean Validation API. By validation, we can validate a regular expression, a number, length, and other related input. It is also possible to offer some custom validations.

This API requires implementation because it is a type of specification. Hence, it makes use of the Hibernate Validator. Usually, the following annotations are a recurring theme in validation.

1. **@Min** – It calculates and ensures that a given number is equal to or greater than a provided value.
2. **@Max** – It calculates and ensures that a given number is equal to or less than a provided value.
3. **@NotNull** – It calculates and ensures that no null value is possible for a field.
4. **@Pattern** – It calculates and ensures that the provided regular expression is followed by the sequence.

Let us work on an example which contains a form with basic input fields. We would use the “\*” sign to represent mandatory fields that must be filled in or else the form causes an error. Begin by adding the required dependencies in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
Now, generate a bean class like the following.
import javax.validation.constraints.Size;

public class Student {
    private String sname;
    @Size(min=1,message="required")
    private String pwd;

    public String getSname() {
        return sname;
    }
    public void setSname(String sname) {
        this.sname = sname;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
}
```

For your controller class, write the following code where the `@Valid` annotation implements rules for validation to the given object while the interface “`BindingResult`” stores the validation’s output.

```
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotationModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class StudentController {
    @RequestMapping("/hello")
    public String show(Model md)
    {
        md.addAttribute("stu", new Student());
        return "vp";
    }
    @RequestMapping("/helloagain")
    public String subfrm( @Valid @ModelAttribute("stu") Student s, BindingResult br)
    {
        if(br.hasErrors())
        {
            return "vp";
        }
        else
        {
            return "last";
        }
    }
}
```

In the `web.xml` file, adjust the following settings for the controller.

```
<?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">
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

Now specify the bean,

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!--Add functionality to scan component -->
    <context:component-scan base-package="com.ith" />
    <!--Add functionality for formatting, conversion, and validation -->
    <mvc:annotation-driven/>
    <!--Specify the view resolver for Spring MVC -->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalRe-
sourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
</beans>
```

For the request page, generate an index.jsp file and place the following code.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
<a href="hello">Hit This Link</a>
</body>
</html>
```

For the view components, generate the vp.jsp page with the following code.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<style>
.error{color:blue}
</style>
</head>
<body>
<form:form action="hiuser" modelAttribute="stu">
Student Name: <form:input path="sname"/> <br><br>
Password(*) : <form:password path="pwd"/>
<form:errors path="pass" cssClass="error"/><br><br>
<input type="submit" value="submit">
</form:form>
</body>
</html>
```

And finally, add a last.jsp page:

```
<html>
<body>
Student Name: ${stu.sname} <br><br>
Password: ${stu.pwd}
</body>
</html>
```

Run this application. The user first finds a hyperlink which redirects to a web page. Afterward, the name and password details of the user are typed. However, this time the “password” field is displayed as “required” when you do not enter it. After typing the password, the screen shows your name along with your password.

## 20.16 | Validation with Regular Expression

In Spring MVC, we can use validation of user input with a specific technique, which is also known as *regular expression*. In order to utilize the validation for regular expression, you have to use the @Pattern annotation. In such a strategy, the regexp attribute is used with the required expression along with our annotation.

Add dependencies in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jasper</artifactId>
    <version>9.0.12</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>
```

Now make the Student.java class which is our bean class.

```
import javax.validation.constraints.Pattern;
public class Student {
    private String sname;
    @Pattern(regexp="^[a-zA-Z0-9]{4}", message="The length should be at least 4")
    private String pwd;

    public String getSname() {
        return sname;
    }
    public void setSname(String sname) {
        this.sname = sname;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
}
```

For the controller class, write the following code.

```
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class StudentController {
    @RequestMapping("/hello")
    public String show(Model md)
    {
        md.addAttribute("stu", new Student());
        return "vp";
    }
    @RequestMapping("/helloagain")
    public String subfrm(@Valid @ModelAttribute("stu") Student s BindingResult br)
    {
        if(br.hasErrors())
        {
            return "vp";
        }
        else
        {
            return "last";
        }
    }
}
```

In the web.xml file, write the following code.

```

<?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/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>

```

Now, for the xml file, specify the bean like this in the spring-servlet file.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd
           http://www.springframework.org/schema/mvc
           http://www.springframework.org/schema/mvc/spring-mvc.xsd">
    <!-- Add functionality to scan component-->
    <context:component-scan base-package="com.ith" />
    <!-- Add functionality for validation, formatting, and conversion -->
    <mvc:annotation-driven/>
    <!--Specify the view resolver in Spring MVC -->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property>
        <property name="suffix" value=".jsp"></property>
    </bean>
</beans>

```

Now, for the initial request, create an index.jsp page and add the following code.

```

index.jsp
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
<a href="hello">Hit This Link To Go Forward!</a>
</body>
</html>

```

To show the view components, create a vp.jsp page and add the following.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<style>
.error{color:blue}
</style>
</head>
<body>
<form:form action="hiuser" modelAttribute="stu">
Student Name: <form:input path="name"/> <br><br>
Password(*) : <form:password path="pass"/>
<form:errors path="pass" cssClass="error"/><br><br>
<input type="submit" value="submit">
</form:form>
</body>
</html>
```

Finally, create one more, last.jsp and add the following.

```
<html>
<body>
Username: ${stu.sname} <br><br>
Password: ${stu.pwd}
</body>
</html>
```

When the application is run, a click leads the user to a new screen which takes the username credential. If the password is shorter than 4 characters, then there is a warning text for it. On typing the right password, the application saves it and proceeds.

## 20.17 | Validation with Numbers

In Spring MVC, we can use validation of user input with a set of numbers. In order to utilize the validation for regular expression, you have to use @Min annotation and @Max annotation. The input has to be more than its value for the former, whereas the input has to lesser than its value for the latter.

Add dependencies in the pom.xml file.

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring-webmvc</artifactId>
<version>5.1.1.RELEASE</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.tomcat/tomcat-jasper -->
<dependency>
<groupId>org.apache.tomcat</groupId>
<artifactId>tomcat-jasper</artifactId>
<version>9.0.12</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/javax.servlet-api -->
<dependency>
<groupId>javax.servlet</groupId>
```

```

<artifactId>servlet-api</artifactId>
<version>3.0-alpha-1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/javax.servlet/jstl -->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.hibernate.validator/hibernate-validator -->
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.13.Final</version>
</dependency>

```

Now make the Student.java class which is our bean class.

```

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.Size;

public class Student {
    private String name;
    @Size(min=1,message="required")
    private String pwd;

    @Min(value=50, message="A student must score 50 or more to qualify.")
    @Max(value=100, message="A student cannot score equal or less than 100.")
    private int marks;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPwd() {
        return pwd;
    }
    public void setPwd(String pwd) {
        this.pwd = pwd;
    }
    public int getMarks() {
        return marks;
    }
    public void setMarks(int marks) {
        this.marks = marks;
    }
}

```

For your controller class, type the following code:

```
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class StudentController {
    @RequestMapping("/hiuser")
    public String show(Model md)
    {
        md.addAttribute("stu", new Student());
        return "vp";
    }
    @RequestMapping("/hiagain")
    public String subfrm( @Valid @ModelAttribute("stu") Student s, BindingResult br)
    {
        if(br.hasErrors())
        {
            return "vp";
        }
        else
        {
            return "last";
        }
    }
}
```

In the web.xml file, add the following:

```
<?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/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
    <display-name>SpringMVC</display-name>
    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

Specify a bean in the XML file. Then for the request page, add the following in the index.jsp.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<body>
<a href="hiuser">Hit this link!</a>
</body>
</html>
```

Now, to generate the view components, write the following:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<html>
<head>
<style>
.error{color:blue}
</style>
</head>
<body>
<form:form action="hiagain" modelAttribute="student">
Name: <form:input path="name"/> <br><br>
Password: <form:password path="pass"/>
<form:errors path="pass" cssClass="error"/><br><br>
Marks: <form:input path="marks"/>
<form:errors path="marks" cssClass="error"/><br><br>
<input type="submit" value="submit">
</form:form>
</body>
</html>
```

In the last.jsp, type the following code:

```
<html>
<body>
Username: ${param.name} <br>
Password: ${param.pwd} <br>
Age: ${param.marks } <br>
</body>
</html>
```

This code would ensure that only students with marks in the range of 50–100 can submit their details.

## Summary

Spring MVC remains one of the leading Java frameworks in the industry today. If you plan to learn web development for educational purposes or to begin a career, then it is an excellent choice. From the government sector to the banking sector, Java Spring MVC is used in a wide range of industries, especially in the enterprises.

In this chapter, we have learned the following concepts:

1. Spring Framework and Spring MVC.
2. Dependency Injection, Inversion of Control, and aspect-oriented programming.
3. Spring architecture including core module, web module, Bean module, and context module.
4. Bean scope such as singleton, prototype, and session.
5. Spring MVC concepts.
6. DispatcherServlet, Context Hierarchy, Special Bean Types, etc.
7. Spring Bean Processing, Interception, Chain of Resolver.
8. View Resolution, Multiple Controller, etc.
9. Various examples to understand concepts such as Model Interface and RequestParam.

In Chapter 21, we will learn about a popular ORM tool known as Hibernate. We will learn how to use ORM tools such as annotations, inheritance mapping, and class to table mapping.

## Multiple-Choice Questions

---

1. Which one of the following components intercepts all requests in a Spring MVC application?
  - (a) DispatcherServlet
  - (b) ControllerServlet
  - (c) FilterDispatcher
  - (d) None of the above
2. Which one of the following components is utilized to map a request to a method of a controller?
  - (a) URL Mapper
  - (b) RequestResolver
  - (c) RequestMapper
  - (d) RequestMapping
3. If we want to apply custom validation, which one of the following interfaces is required to be implemented?
  - (a) Validatable
4. Is it possible to generate multiple controllers in Spring MVC at the same time?
  - (a) Yes
  - (b) No
5. \_\_\_\_\_ uses the DispatcherServlet for invoking handlers which are mapped with a request.
  - (a) HandlerMapping
  - (b) HandlerAdapter
  - (c) LocalResolver
  - (d) FlashMapManager

## Review Questions

---

1. How do you define Spring Framework?
2. What are the core modules of Spring Framework?
3. What is Dependency Injection?
4. What are the benefits of using Dependency Injection?
5. What is Inversion of Control?
6. How does Spring use Inversion of Control? What are the benefits?
7. What is aspect-oriented programming? How do we use it?
8. What are Bean Scopes?
9. What is the default Bean Scope?
10. When do we use Prototype Scope?
11. What is Spring MVC?
12. What is Dispatcher Servlet?
13. How does Autowiring work?
14. How does constructor-based Dependency Injection work?
15. How does setter-based Dependency Injection work?
16. How does field-based Dependency Injection work?
17. Which type of Dependency Injection is better to use? Why?
18. What types of advice are there? Explain in detail.
19. What is Pointcut?
20. What is Join Point?
21. What is the difference between Pointcut and Join Point?
22. What is Target object?
23. What is AOP Proxy?
24. What modules are there in the Data Access/Integration layer?
25. What is the use of Messaging Module?
26. What is context hierarchy?
27. What is Chain of Resolvers? How does it work?
28. What is view resolution?
29. What is the difference between @RequestParam and @PathVariable?

## Exercises

---

1. Setup an environment either using Eclipse or Spring Tool Suite and create a Spring MVC project. Create three views, first to capture student data for registration using FormTag Library, second to display data and third home page to show a school page.
2. Extend this first exercise to add validations on the registration form.

## Project Idea

---

Create a Hotel Booking application using Spring MVC. This app should have frontend view to search for hotels based on users' criteria such as star ratings, reviews, and price. And a feature to sort results based on price or ratings. It should also

have front end view for admin to add hotel information. Use HTML, CSS, JQuery, and Bootstrap for the front end and use Spring MVC on the back end side.

## Recommended Readings

---

1. Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho. 2017. *Pro Spring 5: An In-Depth Guide to the Spring Framework and its Tools*. Apress: New York
2. Ranga Karanam. 2017. *Mastering Spring 5.0*. Packt: Birmingham
3. Tejaswini Mandar Jog. 2017. *Learning Spring 5.0: Build enterprise grade applications using Spring MVC, ORM Hibernate and RESTful APIs*. Packt: Birmingham



# Introduction to Hibernate

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- ORM tools.
- Hibernate and how to use Hibernate.
- How to configure database and connect to it.
- Various useful annotations.
- Inheritance mapping.
- How to map Class and Table.
- How to map with set.
- Transactions and caching.
- Hibernate Query Language.
- All about named query.
- Spring Integration.

## 21.1 | Introduction

Hibernate is one of the most popular and common technologies in Java related projects. It is a framework which acts as an intermediary between a Java application and a database. All the low-level complex work associated with SQL is addressed by Hibernate. Through the addition of Hibernate, the number of lines of code for JDBC can be greatly decreased. The key offering of Hibernate is object-relational mapping (ORM).

What is ORM? As a developer, you have to specify in Hibernate about how an object in your application is mapped to the stored data in the DB. This means that it would map your Java class to any table in the database.

For example, if there is a Java class Employee with four fields:

1. id(int)
2. name(String)
3. designation(String)
4. department(String)

And you have a database table known as Employee.

1. id (INT)
2. name(VARCHAR(45))
3. designation(VARCHAR(45))
4. department(VARCHAR(45))

Then what Hibernate does is that it ensures that this class and table are mapped appropriately. For instance, it can use the object to change the designation of an employee in the table. To generate a Java object in Hibernate, you can write the following code.

```
Employee emp = new Employee ("Albert", "developer", "IT");
```

In order to save it in the DB, you can write the following.

```
int infoId = (Integer) session.save(emp);
```

By doing this, Hibernate is simply running the SQL insert query on the back and saving the data in the DB table. To get or retrieve an object through Hibernate, you can write the following.

```
Employee emp = session.get(Employee.class, infoId);
```

In this code the `session.get` method searches for the class of the object and also takes the primary id of the object (i.e., "infoId"). On the back, Hibernate processes this information by going to the DB and looking for a column with this primary ID.

Now, what to do if you require all of the "Employee" objects? This can be done through Hibernate's support for query. To do this, you can write the following.

```
Query q = session.createQuery("from Employee");
List<Employee> employees = query.list();
```

What the above code does is that it goes in the database table with the name "Employee" and then list all its entries. This is Hibernate Query Language (HQL) at work which would use explain later in our tutorial.

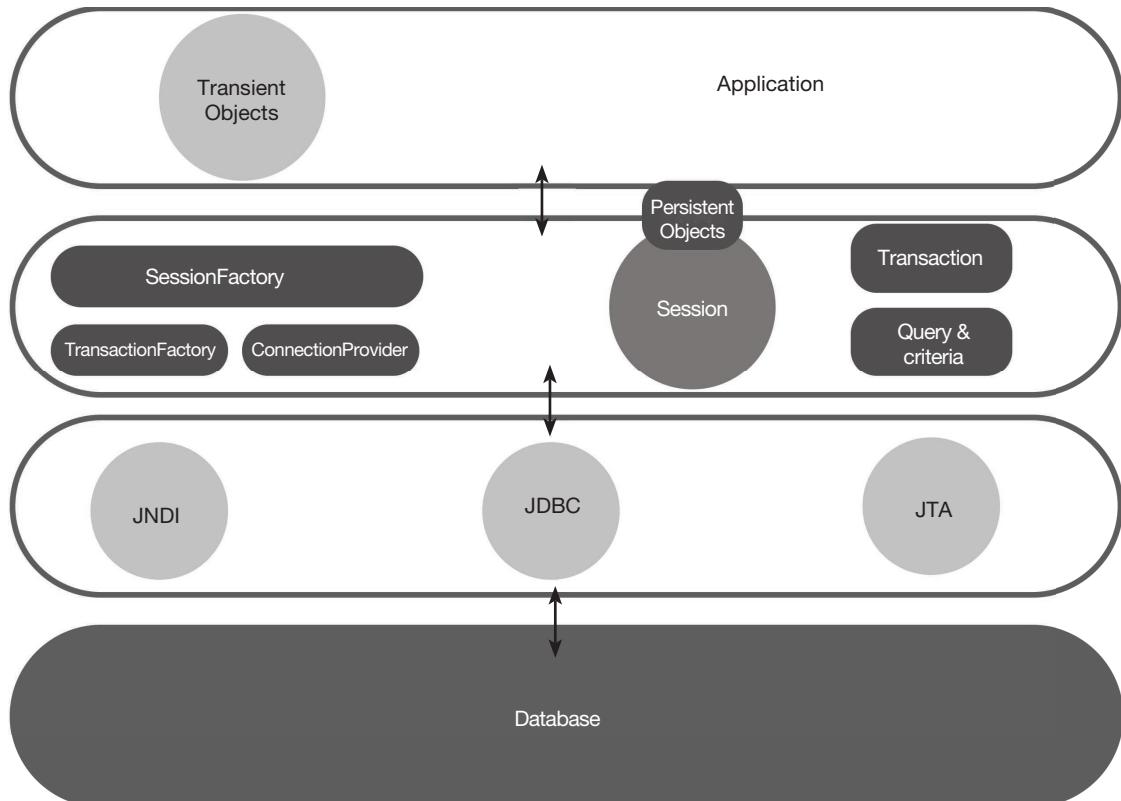
### QUICK CHALLENGE

Create a chart to show the differences between JDBC approach and Hibernate approach. Also list the advantages and disadvantages of using both.

## 21.2 | Architecture



Hibernate uses a layered architecture as shown in Figure 21.1. It allows a user to work without explicitly knowing the underlying APIs. Hibernate takes advantage of the configuration and database data which is used to offer persistence services to the application. Hibernate uses JNDI, JDBC, JTA, Java Naming, and other existing Java APIs. It is used to take advantage of a basic abstraction level for relational databases. This means that it is supported by most of the databases.



**Figure 21.1** Hibernate architecture.

## 21.2.1 Configuration Object

The first Hibernate object in a Hibernate application is called *configuration object*. It is generated during application initialization. It has details of properties and configuration file. This object has two main components:

1. **Database connection:** This is managed by multiple configuration files. These files are hibernate.cfg.xml and hibernate.properties.
2. **Class mapping setup:** This generates the connection between the DB tables and the Java classes.



How can you connect multiple ORM tools to Spring MVC?

## 21.2.2 Session

To establish a physical connection with a database, you need a session. The object of session is lightweight. It is used every time a DB interaction is required. Session objects are used to retrieve persistent objects. The session objects are not allowed to remain open for long periods of time as they are not deemed thread safe. Hence, they are created and destroyed according to the application requirements. The main objective of the session object is to perform read, create, and remove operations on related mapped entity classes instances.

## 21.2.3 Methods

1. **Transaction beginTransaction():** It starts a unit of work and forwards the relevant Transaction object.
2. **Void cancelQuery():** It terminates the current query execution.
3. **void clear():** It fully clears the session.
4. **connection close():** It ends the session either by cleaning or releasing the connections related to JDBC.
5. **criteria createCriteria(Class persistentClass):** It generates a Criteria instance for a superclass of a specified entity class.
6. **CriteriaCreateCriteria (String entityName):** It generates a fresh Criteria instance to help with the entity name.
7. **Serializable getIdentifier (Object object) :** It is used for a given entity's identifier value.



What is the use of Transaction Management in Hibernate?

## 21.2.4 Hibernate Criteria

HQL is mostly used to query the database and receive the results. Usually, it is not preferred for deleting or updating values as then it means that the developer has to manage the table associations.

Hibernate Criteria API offers an object-oriented approach, which can be used to query the DB and get the output. It is important to note that the Hibernate Criteria query cannot be used to delete or update Data Definition Language (DDL) statements or other queries. Instead, it is only used for getting DB results through the use of object-oriented model. This API allows the following:

1. Hibernate Criteria API offers Projection, which can be used for min(), max(), sum(), and other aggregate functions.
2. It uses “ProjectList” for getting the specified columns.
3. Hibernate Criteria is used with join queries where it joins multiple tables.
4. Results with conditions can also be fetched by the Hibernate Criteria API.
5. It also offers the addOrder() for ordering the output.



List the differences between HQL and SQL languages.

## 21.2.5 Cache

Hibernate Cache is valuable in providing quick performance for applications. It helps to decrease DB queries and thus decreases the application throughput. There are three types of Cache.

1. **First level cache:** This cache is linked with the session object. By default, it is enabled and cannot be turned off. Objects which are cache through such a session are invisible for other sessions. After a session is terminate, all the cache objects are removed.
2. **Second level cache:** By default, it is disabled. To enable it, you have to modify the configuration. Infinispan and EHCache offer the Hibernate second level cache.
3. **Query cache:** Hibernate is used also for caching a query's result set. It does not cache entities' state cache. It is used for caching only the identifier values and the value type results. Hence, it is used with the second-level cache.



How will application behave if no Cache is used?

## 21.3 | Installation and Configuration



In order to use hibernate, we need to first install database, configure and setup database. In the following sections, we will be installing database, testing database connection using JDBC, and configuring hibernate.

### 21.3.1 Database

In this chapter, we are using MySQL database for the Hibernate example. Install the latest MySQL Community Server. When you are done installing it, go to Start and search for "Workbench". Open it and connect it to the server instance.

In MySQL, we have created a user "hbstudent", which contains the following columns.

1. Id
2. first\_name
3. last\_name
4. email

#### QUICK CHALLENGE

List all the databases that can work with Hibernate.

### 21.3.2 Eclipse

In Eclipse, create a new "Java Project", we have named ours as "hib". When the project is generated, right click on it and then go to New → Folder. Name the folder as "lib". This folder would be used to store JDBC and Hibernate files. Now, go to Hibernate's official website. Check the latest Hibernate ORM release and download it. After extracting the folder, go in the "required" folder and copy all the files. Come back to Eclipse and paste it in the "lib" folder.

Afterward, open this website and download the latest Connector. After extracting it, copy the JAR file (named like mysql-connector-java) and then paste it in the "lib" folder.

Finally, right click on the project and go to → Properties → Java Build Path → Libraries. Choose "Add Jars" and go to your project and highlight all recently added JAR files. Apply the changes and close the Properties.

### 21.3.3 JDBC

Before moving forward, it is important to test Java Database Connectivity (JDBC). Create a sample package in the project (we have named it "com.hib.jdbc") and then add a class in it with the following code.

```

import java.sql.Connection;
import java.sql.DriverManager;
public class exampleJDBC {
    public static void main(String args[]) {
        String jdbcUrl = "jdbc:mysql://localhost:3306/hb_student_tracker?useSSL=false";
        String user = "hbstudent";
        String pass = "hbstudent";
        try {
            System.out.println("Attempting to connect to DB: " + jdbcUrl);
            Connection myConn = DriverManager.getConnection(jdbcUrl, user, pass);
            System.out.println("Congratulations! The Connection Is Successful!");
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
    }
}

```

If everything is done correctly, then we get the following output.

```

Attempting to connect to DB:
jdbc:mysql://localhost:3306/hb_student_tracker?useSSL=false
Congratulations! The Connection Is Successful!

```

Till now, we have set up and configured everything required to begin with Hibernate. Now to configure Hibernate, we must first have to create a configuration file. This file instructs Hibernate about the information required to establish a connection with the database. It would contain the JDBC URL, id, password, and other pieces of information. Right click on the “src” of the project and then create a new file with the name “hibernate.cfg.xml”. Copy the following code in the file.

```

<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- JDBC Database connection settings -->
        <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/hb_student_tracker?
useSSL=false&serverTimezone=UTC</property>
        <property name="connection.username">hbstudent</property>
        <property name="connection.password">hbstudent</property>
        <!-- JDBC connection pool settings ... using built-in test pool -->
        <property name="connection.pool_size">1</property>
        <!-- Select our SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- Echo the SQL to stdout -->
        <property name="show_sql">true</property>
        <!-- Set the current session context -->
        <property name="current_session_context_class">thread</property>
    </session-factory>
</hibernate-configuration>

```

### 21.3.4 Annotations

In Hibernate, there is a term known as “Entity Class”. It is just a plain old Java object (POJO) which is mapped with a table in the database. This mapping can be done through two options. You can either use the older approach to generate XML configuration files or you can adopt the modern approach to use Annotations.

In Annotations, you have to first map a Java class to a table in the DB and then map its fields to the columns of that DB. To work with Annotations, create a new package in your project and a “Student” class in it. Add the following code in the class.

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name="student")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="id")
    private int id;

    @Column(name="first_name")
    private String firstName;

    @Column(name="last_name")
    private String lastName;

    @Column(name="email")
    private String email;

    public Student() { }

    public Student(String firstName, String lastName, String email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastname() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    @Override
    public String toString() {
        return "Student [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName
+ ", email=" + email + "]";
    }
}
```

**QUICK  
CHALLENGE**

Create a list of all the important annotations.

## 21.4 | Java Objects in Hibernate

In Hibernate, we are going to use a class called SessionFactory. This is the class which goes through the configuration file. It is also responsible for the generation of the session objects. Session factory is generated once and is then used several times in the application. When this class is generated, it then manufactures “sessions”.

The “session” class is the wrapper class for JDBC. It is this object which is required for the storage and retrieval of objects. Bear in mind that it has a short lifespan.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import com.hib.hibernate.demo.entity.Student;
public class CreateStudentDemo {
    public static void main(String[] args) {
        SessionFactory factory = new Configuration().configure("hibernate.cfg.xml") .
addAnnotatedClass(Student.class).buildSessionFactory();
        Session session = factory.getCurrentSession();
        try {
            System.out.println("Generate an object for a student");
            Student tempStudent = new Student("Robert", "William", "robw123@abc.com");
            session.beginTransaction();
            System.out.println("the student object is being saved");
            session.save(tempStudent);
            session.getTransaction().commit();
            System.out.println("Completed!");
        }
        finally {
            factory.close();
        }
    }
}
```

Now, open your MySQL table and you will see a new row in your table. Change the values in the above code and run it again. Now you will see a new record in the table.



How can Hibernate manage database security?

### 21.4.1 Primary Key

Before going further, you must have a basic understanding of primary key (PK). PK is a DB component which is applicable on a column in a table. The main purpose of putting PK on a column is to establish a relationship in relational database. Bear in mind that PK is always unique; this means that you can only assign those columns as PK which cannot have repeated values. For instance, in an employee table, the employee numbers which are unique can be used as PK. It is also important to note that it cannot store NULL values. For instance, if there is an employee “Josh” then to update its data, its employee number “04” can be used in queries where no other employee carries the same employee number. To assign a column as PK, simply write the following line in the SQL query.

Primary Key ("emp\_no")

In Java, you can use GenerationType.AUTO to choose any suitable technique for a specific DB to generate an ID. By using the GenerationType.Identity, primary keys can be assigned with respect to the identity column. The GenerationType.SEQUENCE class processes primary keys according to a DB sequence. Finally, the GenerationType.TABLE processes primary keys with respect to an underlying table in the DB for maintaining uniqueness.

Go to your MySQL Workbench and right click on the table → Alter Table. You can see your columns and the primary key. In our example, we have turned on auto-increment so your AI column will also be ticked.

## 21.5 | Inheritance Mapping



There are three types of inheritance mapping techniques in Hibernate.

### 21.5.1 Table Per Hierarchy

In this mapping, you require `@Inheritance(strategy=InheritanceType.Single_TABLE)`, `@DiscriminatorColumn`, and `@DiscriminatorValue` annotations. This mapping requires only a single table. One more column is generated in the DB's table for calling the class. This table is referred to as the discriminator column. For example, first generate a persistent class which embodies inheritance. To do this, we have produced three classes. First, we have a `Student.java` class.

```
package com.hib.example;
import javax.persistence.*;
@Entity
@Table(name = "Jones")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="student")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "id")
    private int id;
    @Column(name = "name")
    private String name;
    //setters and getters
}
Our second class is the "fulltime_student.java" class.
package com.hib.example;
import javax.persistence.*;

@Entity
@DiscriminatorValue("fulltimestudent")
public class FulltimeStudent extends Student{

    @Column(name="marks")
    private float marks;

    @Column(name="coursenumber")
    private int coursenumber;

    //setters and getters
}
```

Our final class is the `parttime_student.java` class.

Now, in the project go in the source of the `pom.xml` file and add the following dependencies. Make sure that they are placed within the `<dependencies>` tag.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.3.1.Final</version>
</dependency>

<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>10.2.0.4.0</version>
</dependency>
```

Now, you have to open your hibernate.cfg.xml file and add the list of entity classes.

```
?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-5.3.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">system</property>
        <property name="connection.password">jtp</property>
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <mapping class="com.ith.mypackage.Student"/>
        <mapping class="com.ith.mypackage.Fulltime_Student"/>
        <mapping class="com.ith.mypackage.Parttime_Student"/>
    </session-factory>
</hibernate-configuration>
```

However, in order to store persistent object, you will require a class. To do this, create a “StoreTest” Java class.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreTest {
    public static void main(String args[])
    {
        StandardServiceRegistry ssr = new StandardServiceRegistryBuilder().configure
        ("hibernate.cfg.xml").build();
        Metadata meta = new MetadataSources(ssr).getMetadataBuilder().build();
        SessionFactory factory=meta.getSessionFactoryBuilder().build();
        Session session=factory.openSession();
        Transaction t=session.beginTransaction();

        Employee e1=new Employee();
        e1.setName("Gaurav Chawla");

        Regular_Employee e2=new Regular_Employee();
        e2.setName("Vivek Kumar");
        e2.setSalary(50000);
        e2.setBonus(5);

        Contract_Employee e3=new Contract_Employee();
        e3.setName("Arjun Kumar");
        e3.setPay_per_hour(1000);
        e3.setContract_duration("15 hours");

        session.persist(e1);
        session.persist(e2);
        session.persist(e3);

        t.commit();
        session.close();
        System.out.println("success");
    }
}
```

Run this class and you will see the list of students with their names and other details.

## 21.5.2 Table Per Concrete Class

In Table per Concrete class, a table is generated for each class. This means that in the DB, there is no table which has nullable values. On the other hand, this gives rise to duplication of columns.

In such a mapping, the `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)` annotation is used in the parent class, while the `@AttributeOverrides` annotation is used in any of the subclasses. The former annotation defines the implementation of Table per Concrete class technique while the latter annotation specifies that the attributes of the parent class would be overridden.

First, generate persistent classes for inheritance.

```
import javax.persistence.*;

@Entity
@Table(name = "employee")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)

    @Column(name = "eid")
    private int eid;

    @Column(name = "e_name")
    private String ename;

    //setters and getters
}
```

Now, generate a class for permanent employees.

```
import javax.persistence.*;

@Entity
@Table(name="permanentemployees102")
@AttributeOverrides({
    @AttributeOverride(name="eid", column=@Column(name="eid")),
    @AttributeOverride(name="ename", column=@Column(name="ename"))
})
public class PermanentEmployee extends Employee{

    @Column(name="pay")
    private float pay;

    @Column(name="leaves")
    private int leaves;

    //setters and getters
}
```

For temporary employees, generate the following class.

```

import javax.persistence.*;
@Entity
@Table(name="temporaryemployees")
@AttributeOverrides({
    @AttributeOverride(name="eid", column=@Column(name="eid")),
    @AttributeOverride(name="ename", column=@Column(name="ename"))
})
public class temporary_Employee extends Employee{
    @Column(name="hourly_wage")
    private float hourly_wage;

    @Column(name="services_type ")
    private String services_type;

    public float gethourly_wage() {
        return hourly_wage;
    }
    public void sethourly_wage(float hourlywage) {
        hourly_wage = hourlywage;
    }
    public String getservices_type() {
        return services_type;
    }
    public void setservices_type(String servicestype) {
        services_type = servicestype;
    }
}

```

Now, in the project go in the source of the pom.xml file and add the following dependencies. Make sure that they are placed within the <dependencies> tag.

```

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.3.1.Final</version>
</dependency>

<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>10.2.0.4.0</version>
</dependency>

In the hibernate.cfg.xml file, apply the following mapping.
?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<!-- Generated by MyEclipse Hibernate Tools. -->
<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">system</property>
        <property name="connection.password">abc</property>
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <mapping class="com.ith.examples.Employee"/>
        <mapping class="com.ith.examples.PermanentEmployee"/>
        <mapping class="com.ith.examples.Temporary_Employee"/>
    </session-factory>
</hibernate-configuration>

```

Now generate a class `StoreData` which can save all these objects related to employee tables.

```

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreData {
    public static void main(String[] args) {
        StandardServiceRegistry reg=new StandardServiceRegistryBuilder().configure
        ("hibernate.cfg.xml").build();
        Metadata meta=new MetadataSources(reg).getMetadataBuilder().build();
        SessionFactory factory=meta.getSessionFactoryBuilder().build();
        Session session=factory.openSession();
        Transaction trans=session.beginTransaction();

        Employee first=new Employee();
        first.setename("Jack");

        PermanentEmployee second=new PermanentEmployee();
        second.setename("Jones");
        second.setpay(4000);
        second.setleaves(8);

        Temporary_Employee third=new Temporary_Employee();
        third.setename("Bill");
        third.sethourly_wage(2000);
        third.setservices_type("IT");

        session.persist(first);
        session.persist(second);
        session.persist(third);

        trans.commit();
        session.close();
        System.out.println("We have completed the operation successfully");
    }
}

```

### 21.5.3 Table Per Subclass

In table per subclass, tables are generated as persistent classes though they act as primary and foreign keys. This means that it is not possible to have duplicate columns in the table.

In this technique, you require the parent class to have “`@Inheritance(strategy=InheritanceType.JOINED)`” and the subclasses to have `@PrimaryKeyJoinColumn` annotations.

First, we have to generate the persistent classes for inheritance. The first “Employee” class contains the following code.

```

import javax.persistence.*;
@Entity
@Table(name = "employee105")
@Inheritance(strategy=InheritanceType.JOINED)
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "eid")
    private int eid;

    @Column(name = "ename")
    private String ename;

    //setters and getters
}

```

In the second class, “PermanentEmployee” we have the following code.

```
import javax.persistence.*;
@Entity
@Table(name="permanentemployee105")
@PrimaryKeyJoinColumn(name="eid")
public class PermanentEmployee extends Employee{

    @Column(name="pay")
    private float pay;
    @Column(name="leaves")
    private int leaves;

    //setters and getters
}
```

In the third class, we have the following code.

```
import javax.persistence.*;

@Entity
@Table(name="temporaryemployee105")
@PrimaryKeyJoinColumn(name="eid")
public class Temporary_Employee extends Employee{

    @Column(name="hourly_wage")
    private float hourly_wage;

    @Column(name="services_type")
    private String services_type;

    //setters and getters
}
```

Now, in the project go in the source of the pom.xml file and add the following dependencies. Make sure that they are placed within the <dependencies> tag.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.3.1.Final</version>
</dependency>

<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>10.2.0.4.0</version>
</dependency>
```

For the configuration file, do the following.

```

?xml version='1.0' encoding='UTF-8'?
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<!-- Generated by MyEclipse Hibernate Tools. -->
<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">system</property>
        <property name="connection.password">abc</property>
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>

        <mapping class="com.ith.examples.Employee"/>
        <mapping class="com.ith.examples.PermanentEmployee"/>
        <mapping class="com.ith.examples.Temporary_Employee"/>
    </session-factory>
</hibernate-configuration>

```

Now in the final StoreData class, write the following code.

```

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreData {
    public static void main(String[] args) {
        StandardServiceRegistry reg=new StandardServiceRegistryBuilder()
configure("hibernate.cfg.xml").build();
        Metadata meta=new MetadataSources(reg).getMetadataBuilder().build();
        SessionFactory factory=meta.getSessionFactoryBuilder().build();
        Session session=factory.openSession();
        Transaction trans=session.beginTransaction();
        Employee first=new Employee();
        first.setename("Ryan");

        PermanentEmployee second=new PermanentEmployee();
        second.setename("James");
        second.setpay(7000);
        second.setleaves(4);

        Temporary_Employee third=new Temporary_Employee();
        third.setename("Luke");
        third.sethourly_wage(5000);
        third.setservices_type("accounting");

        session.persist(first);
        session.persist(second);
        session.persist(third);

        trans.commit();
        session.close();
        System.out.println("We have completed the operation successfully");
    }
}

```



## 21.6 | Collection Mapping

Hibernate supports mapping of collection elements in the persistent class. Their types include list, set, map, and collection. The persistent class has to be specified like the following:

```
import java.util.List;

public class problem {
    private int id;
    private String pname;
    private List<String> solutions;

    //getters and setters
}
```

There are numerous sub-elements in the <class> elements for mapping collections.

### 21.6.1 Mapping List

When the persistent class contains objects in the “list” format, they can be through either annotation or file. For the above class, we can write.

```
<class name="com.ihrt.problem" table="p100">
    <id name="id">
        <generator class="increment"></generator>
    </id>
    <property name="pname"></property>

    <list name="solutions" table="sol100">
        <key column="pid"></key>
        <index column="type"></index>
        <element column="solution" type="string"></element>
    </list>
</class>
```

Here, the <key> element is required to specify the table’s foreign key with respect to the problem class identifier. To specify the type, the <index> element is required. To specify the collection’s element, <element> is used.

In case collection saves objects of another class, it is necessary to specify the types of relationship one-to-many or many-to-many. In such instances, the persistent class resembles the following.

```
import java.util.List;

public class problem {
    private int id;
    private String pname;
    private List<Solution> solutions;

    //getters and setters
}
import java.util.List;
public class Solution {
    private int id;
    private String solution;
    private String posterName;
    //getters and setters
}
```

The mapping file would be similar to the following.

```
<class name="com.ith.problem" table="p100">
    <id name="id">
        <generator class="increment"></generator>
    </id>
    <property name="pname"></property>
    <list name="solutions" >
        <key column="pid"></key>
        <index column="type"></index>
        <one-to-many class="com.ith.problem"/>
    </list>
</class>
```

The one-to-many relationship here means that a single problem can have multiple solutions.

### 21.6.2 Key Element

As already explained, the key element is required to place foreign keys in tables. These tables are joined. By default, FK element is nullable. For non-nullable FK, it is important to define attributes which are not null like.

```
<key column="pid" not-null="true"></key>
```

### 21.6.3 Collection Mapping with Lists

Generate a persistent class.

```
import java.util.List;
public class Problem {
    private int id;
    private String pname;
    private List<String> solutions;
    //getters and setters
}
```

Now create a configuration file named as problem.hbm.xml and apply the following.

```
?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">

<hibernate-mapping>
    <class name="com.ith.problem" table="p100">
        <id name="id">
            <generator class="increment"></generator>
        </id>
        <property name="pname"></property>
        <list name="solutions" table="sol100">
            <key column="pid"></key>
            <index column="type"></index>
            <element column="solution" type="string"></element>
        </list>
    </class>
</hibernate-mapping>
```

In the configuration file, apply the following.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">user1</property>
        <property name="connection.password">abc</property>
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <mapping resource="problem.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

To save the data of the problem class, use the StoreData class.

```
import java.util.ArrayList;
import org.hibernate.*;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreData {
    public static void main(String[] args) {
        StandardServiceRegistry reg=new StandardServiceRegistryBuilder()
            .configure("hibernate.cfg.xml").build();
        Metadata meta=new MetadataSources(reg).getMetadataBuilder().build();

        SessionFactory fac=meta.getSessionFactoryBuilder().build();
        Session session=fac.openSession();

        Transaction ts=session.beginTransaction();

        ArrayList<String> l1=new ArrayList<String>();
        l1.add("Spring is an ecosystem");
        l1.add("Spring is used to create enterprise web applications");

        ArrayList<String> l2=new ArrayList<String>();
        l2.add("Hibernate is an intermediary between database and applications");
        l2.add("Hibernate is used for managing the dependencies of the application");

        Problem p1=new Problem();
        p1.setPname("What is Spring Framework and why is it used?");
        p1.setSolutions(l1);

        Problem p2=new Problem();
        p2.setPname("What is Hibernate and why is it used?");
        p2.setSolutions(l2);

        session.persist(p1);
        session.persist(p2);

        ts.commit();
        session.close();
        System.out.println("The operation was successful");
    }
}
```

## 21.6.4 Mapping with Set

When there is a set object in the persistent class, then mapping can be done through the mapping file's set element. This element does not need an index element. Set differs from the list as it only contains unique values. Generate a persistent class named "Problem" which has the properties such as pname and solutions.

```
import java.util.Set;
public class Problem {
    private int id;
    private String pname;
    private Set<String> solutions;
    //getters and setters
}
```

Now create a configuration file named problem.hbm.xml and apply the following.

```
?xml version='1.0' encoding='UTF-8'?
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">

<hibernate-mapping>
    <class name="com.ith.problem" table="p1003">
        <id name="id">
            <generator class="increment"></generator>
        </id>
        <property name="pname"></property>
        <set name="solutions" table="sol1003">
            <key column="pid"></key>
            <index column="type"></index>
            <element column="solution" type="string"></element>
        </set>
    </class>
</hibernate-mapping>
```

In the configuration file, apply the following.

```
<?xml version='1.0' encoding='UTF-8'?
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">user1</property>
        <property name="connection.password">abc</property>
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <mapping resource="problem.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

To save the data of the problem class, use the StoreData class.

```

import java.util.HashSet;
import org.hibernate.*;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreData {
    public static void main(String[] args) {
        StandardServiceRegistry reg=new StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
        Metadata meta=new MetadataSources(reg).getMetadataBuilder().build();
        SessionFactory fac=meta.getSessionFactoryBuilder().build();
        Session session=fac.openSession();
        Transaction ts=session.beginTransaction();

        HashSet<String> s1=new HashSet<String>();
        s1.add("Linked list is a data structure");
        s1.add("Linked list is good for memory purposes");

        HashSet<String> s2=new HashSet<String>();
        s2.add("In stack data structure, the last data to be inserted is processed first");
        s2.add("Stack data structure is based upon the real world concept of stacks like book stacks");

        Problem p1=new Problem();
        p1.setPname("What is a linked list and why is it used?");
        p1.setSolutions(s1);

        Problem p2=new Problem();
        p2.setPname("What is Stack data structure and where did the idea come from?");
        p2.setSolutions(s2);

        session.persist(p1);
        session.persist(p2);

        ts.commit();
        session.close();
        System.out.println("The operation was successful");
    }
}

```

## 21.7 | Mapping with Map

With Hibernate, it is possible to “map” elements of Map along with relational database management systems. The map is a collection which uses indexes on its back. For the problem class, write the following code.

```

import java.util.Map;

public class Problem {
    private int id;
    private String pname,username;
    private Map<String, String> solutions;

    public Problem() {}
    public Problem(String pname, String username, Map<String, String> solutions) {
        super();
        this.pname = pname;
        this.username = username;
        this.solutions = solutions;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getPname() {
        return pname;
    }
    public void setPname(String pname) {
        this.pname = pname;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public Map<String, String> getSolutions() {
        return solutions;
    }
    public void setSolutions(Map<String, String> solutions) {
        this.solutions = solutions;
    }
}

```

For the problem.hbm.xml file, apply the following.

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">
<hibernate-mapping>
    <class name="com.ith.problem" table="problem800">
        <id name="id">
            <generator class="native"></generator>
        </id>
        <property name="name"></property>
        <property name="username"></property>

        <map name="solutions" table="solution800" cascade="all">
            <key column="problemid"></key>
            <index column="solution" type="string"></index>
            <element column="username" type="string"></element>
        </map>
    </class>
</hibernate-mapping>

```

For the hibernate.cfg.xml file, apply the following.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hbm2ddl.auto">update</property>
    <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
    <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
    <property name="connection.username">user1</property>
    <property name="connection.password">abc</property>
    <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver
  </property>
    <mapping resource="problem.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

To save the data and run the application, generate the “StoreData” class and add the following.

```
Import java.util.HashMap;
import org.hibernate.*;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class StoreData {
  public static void main(String[] args) {
    StandardServiceRegistry reg=new StandardServiceRegistryBuilder().configure
("hibernate.cfg.xml").build();
    Metadata meta=new MetadataSources(reg).getMetadataBuilder().build();

    SessionFactory fac=meta.getSessionFactoryBuilder().build();
    Session session=fac.openSession();

    Transaction ts=session.beginTransaction();
    HashMap<String, String> m1=new HashMap<String, String>();
    m1.put("Linked list is a data structure", "Adam");
    m1.put("Linked list is good for memory purposes", "Keith");

    HashMap<String, String> m2=new HashMap<String, String>();
    m2.put("In stack data structure, the last data to be inserted is processed first",
"Daniel");
    m2.put("Stack data structure is based upon the real world concept of stacks like
book stacks", "Daniel");
    problem p1=new problem("What is Linked List and why is it used?", "Scully", m1);
    problem p2=new problem("What is Stack?", "Simon", m2);
    session.persist(p1);
    session.persist(p2);

    ts.commit();
    session.close();
    System.out.println("The operation was successful");
  }
}
```

## 21.7.1 One-to-Many

When the persistent class contains an object in the form of list with the reference for entity, then mapping is performed through the one-to-many association. In these scenarios, one problem can have multiple solutions where each problem has its own details. Such a persistent class would have the following code.

```
import java.util.List;
public class Problem {
    private int id;
    private String pname; // name of the problem
    private List<Solution> solutions;
}
```

On the other hand, the Solution class can contain details like the name of the solution, poster etc.

```
public class Solution {
    private int id;
    private String solutionname;
    private String poster; // the one who posted the solution
}
```

In the Problem class, there are entity's references in the list object for which you required to apply one-to-many association.

In the mapping file, apply the following settings.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 5.3//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-5.3.dtd">

<hibernate-mapping>
    <class name="com.ith.Problem" table="p601">
        <id name="id">
            <generator class="increment"></generator>
        </id>
        <property name="pname"></property>

        <list name="Solutions" cascade="all">
            <key column="pid"></key>
            <index column="type"></index>
            <one-to-many class="com.ith.Solution"/>
        </list>
    </class>

    <class name="com.ith.Solution" table="sol601">
        <id name="id">
            <generator class="increment"></generator>
        </id>
        <property name="solutionname"></property>
        <property name="poster"></property>
    </class>
</hibernate-mapping>
```

For the configuration file, apply the following settings.

```
?xml version='1.0' encoding='UTF-8'?
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 5.3//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hbm2ddl.auto">update</property>
    <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
    <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
    <property name="connection.username">abc</property>
    <property name="connection.password">xyz</property>
    <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <mapping resource="question.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

To save the data, you can generate the following class.

```
import java.util.ArrayList;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
public class StoreData {
  public static void main(String[] args) {
    StandardServiceRegistry reg= new StandardServiceRegistryBuilder().configure
("hibernate.cfg.xml").build();
    Metadata m=new MetadataSources(reg).getMetadataBuilder().build();
    SessionFactory fact=m.getSessionFactoryBuilder().build();
    Session session=fact.openSession();
    Transaction ts=session.beginTransaction();
    Solution sol1=new Solution();
    sol1.setSolutionname("HTML is a markup language");
    sol1.setPoster("Andrew");

    Solution sol2=new Solution();
    sol2.setSolutionname("HTML is a front-end technology");
    sol2.setPoster("Michael David");
    Solution sol3=new Solution();
    sol3.setSolutionname("Java is used to develop Android applications. It is also used
to create desktop and web applications.");
    sol3.setPoster("Johnny Garcia");
    Solution sol4=new Solution();
    sol4.setSolutionname("Spring Boot minimizes the amount of lines of code which is
usually required in Spring framework. Therefore, it is used for quick projects.");
    sol4.setPoster("Daniel Smith");
    ArrayList<Solution> l1=new ArrayList<Solution>();
    l1.add(sol1);
    l1.add(sol2);
    ArrayList<Solution> l2=new ArrayList<Solution>();
    l2.add(sol3);
    l2.add(sol4);
```

```

        Problem p1=new Problem();
        p1.setPname("What is HTML?");
        p1.setSolutions(l1);

        Problem p2=new Problem();
        p2.setPname("What is SpringBoot and why is this framework used?");
        p2.setSolutions(l2);

        session.persist(p1);
        session.persist(p2);

        ts.commit();
        session.close();
        System.out.println("The operation was completed successfully");
    }
}

```

## 21.7.2 Transaction

A transaction is used to mark a unit of work. When a single step faces failure then the complete process faces failure. Hibernate provides an interface for transaction. In Hibernate, Sessions are used to call transactions. Following are some of the methods of transactions.

1. **void begin():** Initiates a fresh transaction.
2. **void commit():** Puts a stop to the unit of work.
3. **boolean isAlive():** Scans whether the transaction is dead or alive.
4. **void setTimeout(int seconds):** It specifies a time interval when a transaction is initiated due to a call.
5. **void rollback():** Rollsbacks the transaction.

It is recommended that whenever an exception is expected in Hibernate, you should attempt at rollbacks the transaction. This strategy ensures that the resources are free. For instance, consider the following code.

```

Session ssn = null;
Transaction ts = null;
try {
    ssn = sessionFactory.openSession();
    ts = ssn.beginTransaction();
    //code for an action
    ts.commit();
} catch (Exception e) {
    e.printStackTrace();
    ts.rollback();
}
finally {
    ssn.close();
}

```

## 21.8 | Hibernate Query Language

Similar to the database query language SQL, Hibernate has its own query language known as Hibernate Query Language, or HQL in short. HQL is not reliant on the table name of a database. Instead, it makes use of the class name to apply its function. Therefore, it is considered to be independent of database. Hibernate is easy to learn for Java developers. Additionally, it also offers polymorphic queries.



The query interface is an OOP implementation of Hibernate's Query. Query objects can be accessed through the use of `createQuery()` method.

In order to generate data for the existing records, you can write the following code for a persistent class with the name "student". As you can observe, instead of calling a table, HQL is calling for the class through its identifier.

```
Query q = session.createQuery("from Student");//  
List l = q.list();
```

To update the details of a user, you can write the following.

```
Transaction ts = ssn.beginTransaction();  
Query q=session.createQuery("Update update User set name=:n where id=:i");  
q.setParameter("n","Buck");  
q.setParameter("i",222);  
  
int sts=q.executeUpdate();  
System.out.println(sts);  
ts.commit();
```

For deletion, consider the following example.

```
Query q=session.createQuery("delete from Student where id=305");  
q.executeUpdate();
```

You can also apply aggregate functions in HQL. For instance to check the average marks,

```
Query query =session.createQuery("select avg(marks) from Student");  
List<Integer> l=query.list();  
System.out.println(list.get(0));
```

To check the minimum marks for a student, you can write the following.

```
Query query =session.createQuery("select min(marks) from Student");
```

To check the maximum marks for a student, you can write the following.

```
Query query =session.createQuery("select max(marks) from Student");
```

To check the total number of students, you can write the following.

```
Query query =session.createQuery("select count(id) from Student");
```

To check the total marks of students, you can write the following.

```
Query query = session.createQuery("select sum(marks) from Student");
```

### 21.8.1 HCQL

To retrieve records according to the customized filter, Hibernate Criteria Query Language (HCQL) is used. It contains an interface for criteria which is composed of several methods to define set criteria for retrieval of information. For example, you can use HCQL to only see those students who have scored more than 80 marks. Similarly, HCQL can be used to identify only those Students who are in their final year.

To receive all the records for HCQL, you can write the following code.

```
Criteria crt=session.createCriteria(Student.class);  
List l=crt.list();  
To check only those records which exist between the 30th and 40th position, you can  
write the following.  
Criteria crt=session.createCriteria(Student.class);  
crt.setFirstResult(30);  
crt.setMaxResult(40);  
List l=crt.list();
```

To check the records on the basis of marks of students in an ascending arrangement, you can write the following.

```
Criteria crt = session.createCriteria(Student.class);  
crt.addOrder(Order.asc("marks"));  
List l = crt.list()
```

To check records of only those students who have scored more than 75, you can write the following code.

```
Criteria crt = session.createCriteria(Student.class);  
crt.add(Restrictions.gt("marks",75));  
List l = crt.list();
```

To get a specific column through projection, you can write the following piece of code.

```
Criteria crt = session.createCriteria(Student.class);  
crt.setProjection(Projections.property("department"));  
List l = crt.list();
```

### 21.8.2 Hibernate Named Query

The Hibernate named query is a strategy which is used to run queries through a specific name. It is similar to the DB concept of alias names. To run a single query, you can use the @NamedQuery annotation while multiple queries can be handled through @NameQueries.

For instance, consider the following code.

```

import javax.persistence.*;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@NamedQueries(
{
    @NamedQuery(
        name = "findStudentByName",
        query = "from Student s where s.name = :name"
    )
}
)

@Entity
@Table(name="st")
public class Student {
    public String toString(){return id+" "+name+" "+marks+" "+subject;}

    int id;
    String name;
    int marks;
    String subject;
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)

    //getters and setters
}

```

In the configuration file, you can save details pertaining to username, password, driver class, etc.

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 5.3//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-5.3.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hbm2ddl.auto">update</property>
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
        <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
        <property name="connection.username">abc</property>
        <property name="connection.password">xyz</property>
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <mapping class="com.ith.Student"/>
    </session-factory>
</hibernate-configuration>

```

You can use a FetchInfo class to add your named query.

```

import java.util.*;
import javax.persistence.*;
import org.hibernate.*;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class FetchInfo {
    public static void main(String[] args) {
        StandardServiceRegistry reg=new StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();
        Metadata m=new MetadataSources(reg).getMetadataBuilder().build();
        SessionFactory factory=m.getSessionFactoryBuilder().build();
        Session session=factory.openSession();
        TypedQuery q = session.getNamedQuery("findStudentByName");
        q.setParameter("name","Jack");
        List<Student> students=q.getResultList();
        Iterator<Student> i=students.iterator();
        while(i.hasNext()){
            Student s=i.next();
            System.out.println(s);
        }
        session.close();
    }
}

```

## 21.9 | Caching



To enhance an application's performance, Hibernate offers caching which uses the cache to pool an object. This is valuable when the same data requires to be processed continuously. There are two types of cache: first and second level cache.

1. The data of first level cache is stored by the Session object and is configured to be turned on by default. The application as a whole does not have access to this cache.
2. The data of second level cache is stored by the SessionFactory object. Unlike the first level cache, this cache can be accessed by the entire application.

The below example generates a persistent class for Student.

```

import javax.persistence.*;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;
@Entity
@Table(name="s101")
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY)
public class Student {
    @Id
    private int id;
    private String name;
    private float marks;

    public Student() {}
}

```

```

public Student(String name, float marks) {
    super();
    this.name = name;
    this.marks = marks;
}
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public float getMarks() {
    return marks;
}
public void setMarks(float marks) {
    this.marks = marks;
}
}
}

```

In the pom.xml file, add the following dependencies.

```

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.16.Final</version>
</dependency>

<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc14</artifactId>
    <version>10.2.0.4.0</version>
</dependency>

<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>2.10.3</version>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>5.2.16.Final</version>
</dependency>

```

In the configuration file, you have to specify the cache.provider\_class in the property.

```
?xml version='1.0' encoding='UTF-8'?
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 5.2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-5.2.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="show_sql">true</property>
    <property name="hbm2ddl.auto">update</property>
    <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
    <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
    <property name="connection.username">abc</property>
    <property name="connection.password">xyz</property>
    <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
    <property name="cache.use_second_level_cache">true</property>
    <property name="cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
    <mapping class="com.ith.Student"/>
  </session-factory>
</hibernate-configuration>
```

To generate the class that can be used for the retrieval of the persistent object, consider the following code.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.Metadata;
import org.hibernate.boot.MetadataSources;
import org.hibernate.boot.registry.StandardServiceRegistry;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;

public class FetchInfo {
    public static void main(String[] args) {
        StandardServiceRegistry reg=new StandardServiceRegistryBuilder()
        .configure("hibernate.cfg.xml").build();
        Metadata m=new MetadataSources(reg).getMetadataBuilder().build();
        SessionFactory fact=m.getSessionFactoryBuilder().build();
        Session ssn=fact.openSession();
        Student s1=(Student)ssn.load(Student.class,202);
        System.out.println(s1.getId()+" "+s1.getName()+" "+s1.getMarks());
        ssn.close();
        Session ssn2=fact.openSession();
        Employee s2=(Employee)ssn2.load(Employee.class,202);
        System.out.println(s2.getId()+" "+s2.getName()+" "+s2.getSalary());
        ssn2.close();
    }
}
```

## 21.10 | Spring Integration



Spring is one of the most popular Java web frameworks in the industry. Hibernate is commonly used with Spring applications to build enterprise level web projects. Unlike previous cases where it was necessary to use the hibernate.cfg.xml file, this is not required in the case of Spring. Instead, those details would go in the file named applicationContext.xml.

A HibernateTemplate class comes up with the Spring framework, thereby eliminating the need of configuration, session, BuildSessionFactory, and transactions. For instance, without Spring, Hibernate would be required to look like the following code for a student.

```

Configuration conf=new Configuration();
conf.configure("hibernate.cfg.xml");
SessionFactory fact=conf.buildSessionFactory();
Session ssn=fact.openSession();
Transaction ts=ssn.beginTransaction();
Student s1=new Student(222,"Robert",67);
ssn.persist(s1);
ts.commit();
ssn.close();

```

On the other hand, if you use Hibernate's Template which is provided by Spring, then you can minimize the above code into only two lines.

```

Student s1=new Student(222,"Robert",67);
hibernateTemplate.save(s1);

```

For a more detailed example, use any database to generate a table.

```

CREATE TABLE "ST200"
( "ID" NUMBER(10,0) NOT NULL ENABLE,
  "NAME" VARCHAR2(255 CHAR),
  "MARKS" FLOAT(126),
  PRIMARY KEY ("ID") ENABLE
)

```

Now generate a Student class.

```

public class Student {
    private int id;
    private String name;
    private float marks;
}

```

For the mapping file,

```

<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="com.ith.Student" table="ST200">
        <id name="id">
            <generator class="assigned"></generator>
        </id>
        <property name="name"></property>
        <property name="marks"></property>
    </class>
</hibernate-mapping>

```

Use a class which makes use of the `HibernateTemplate`.

```
import org.springframework.orm.hibernate3.HibernateTemplate;
import java.util.*;
public class StudentDao {
    HibernateTemplate t;
    public void setTemplate(HibernateTemplate t) {
        this.t = t;
    }
    // to store the data of student
    public void saveStudent(Student s) {
        t.save(s);
    }
    //to update the data of student
    public void updateStudent(Student s){
        t.update(s);
    }
    //to delete student
    public void deleteStudent(Student s) {
        t.delete(s);
    }
    //to get a student through id
    public Student getById(int id){
        Student s=(Student)t.get(Student.class,id);
        return s;
    }
    //to get all the students
    public List<Student> getStudents(){
        List<Student> l=new ArrayList<Student>();
        l=t.loadAll(Student.class);
        return l;
    }
}
```

In the file for `applicationContext.xml`, you have to add the details of the DB for an object named as `BasicDataSource`. The use of this object can be seen in another object, `LocalSessionFactoryBean`, which holds data related to the `HibernateProperties` and `mappingResources`. This object is required in the class of `HibernateTemplate`. Your `applicationContext.xml` must look like the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/></property>
        <property name="url" value="jdbc:oracle:thin:@localhost:1521:xe"/></property>
        <property name="username" value="abc"/></property>
        <property name="password" value="xyz"/></property> </bean>
    <bean id="mysessionFactory" class="org.springframework.orm.hibernate3.
LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/></property>
        <property name="mappingResources">
            <list>
                <value>student.hbm.xml</value> </list> </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
                <prop key="hibernate.hbm2ddl.auto">update</prop>
                <prop key="hibernate.show_sql">true</prop>
            </props> </property> </bean>
        <bean id="template" class="org.springframework.orm.hibernate3.HibernateTemplate">
            <property name="sessionFactory" ref="mysessionFactory"/></property> </bean>
        <bean id="d" class="com.ith.StudentDao">
            <property name="template" ref="template"/></property> </bean>
        </beans>
```

You can then generate an Insert class which makes use of the StudentDao object and calls out the saveStudent method through the object passing of Student class.

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

public class Insert {
    public static void main(String[] args) {
        Resource res=new ClassPathResource("applicationContext.xml");
        BeanFactory fact=new XmlBeanFactory(res);
        EmployeeDao d=(EmployeeDao)fact.getBean("d");
        Student s=new Student();
        s.setId(102);
        s.setName("Mike");
        s.setMarks(77);
        d.saveStudent(s);
    }
}
```



How do we fetch data in case lazy loading is defined?

## Summary

In the earlier days, software developers had to struggle a lot to link databases with Java applications. They had to consider an extensive list of factors and therefore were often left puzzled to find a solution. As a consequence, a lot of complex code was generated. Luckily, Hibernate's entry to the scene has changed things considerably.

Today, Hibernate has become an integral part of Java applications. Hence, if you plan to use databases heavily in your application, then do not take the hard way. Simply begin learning Hibernate and easily manage your objects and tables with utmost ease.

In this chapter, we have learned the following concepts:

1. How ORM tools work.
2. Use of hibernate in mapping classes with database tables.
3. How inheritance mapping works.
4. Various annotations to map entity relations.
5. How transactions work with Hibernate.
6. How caching works with Hibernate.
7. How to integrate with Spring.
8. How to fetch record with Hibernate Query Language.
9. How to use named query.

In Chapter 22, we will work on the project we have defined in Chapter 2. We have worked on the front end side of it and now we will work on setting up Spring MVC web services.

## Multiple-Choice Questions

1. Which of the following objects is used to generate SessionFactory object in Hibernate?
 

(a) Session	(b) SessionFactory
(c) Transaction	(d) Configuration

2. Which of the following is not a fetching strategy?
    - (a) Sub-select fetching
    - (b) Select fetching
    - (c) Join fetching
    - (d) Dselect fetching
  3. What is the full form of QBC?
    - (a) Query By Column
    - (b) Query By Code
    - (c) Query By Criteria
    - (d) Query By Call
  4. Which of the following is the file where database table configuration is stored?
- (a) .ora
  - (b) .sql
  - (c) .hbm
  - (d) .dbm
5. Which one of the following is not a valid type of cache?
    - (a) First Level
    - (b) Second Level
    - (c) Third Level
    - (d) None of the above

## Review Questions

---

1. What is ORM? What are the benefits of using it?
2. How can Hibernate connect with Spring MVC?
3. How do you define one-to-one relationship?
4. How do you define one-to-many relationship?
5. What are the essential annotations for setting up Hibernate entity on the Spring side?
6. How do you connect database with Hibernate?

## Exercises

---

1. Design a database for managing inventory in a grocery store. Create all the tables and fields for the same and setup hibernate project for the same. Take examples from this chapter.
2. For the above project, create a list of primary and foreign keys that you will map with the variables from the Java class side.

## Project Idea

---

For the Hotel Booking application we have created using Spring MVC in Chapter 20, create a database. Design tables and their relationships based on the model classes and think of the relationship mapping with annotations.

## Recommended Readings

---

1. Christian Bauer, Gavin King, Gary Gregory, and Linda Demichiel. 2017. *Java Persistence with Hibernate, Second Edition*. Manning Publications: New York
2. Joseph B. Ottinger, Jeff Linwood, and Dave Minter. 2016. *Beginning Hibernate: For Hibernate 5*. Apress: New York
3. Yogesh Prajapati and Vishal Ranapariya. 2015. *Java Hibernate Cookbook*. Packt: Birmingham

# Develop Web Services for the APIs

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- How to create a Spring MVC project using Spring Boot.
- How to create the REST web service endpoints.
- Practical use of MVC framework.
- Basic use of Eclipse based IDE like Spring Tool Suite.
- Data Access Object design pattern.
- Useful annotations like @Autowired, @RequestMapping, and @PathVariable.

## 22.1 | Setting up Environment



The foremost requirement for setting up the development environment is installing the Java Development Kit (JDK). For this project, we will use JDK 11. So please visit the official Oracle website to download the latest JDK 11 for your operating system. Once you have installed the JDK 11, you may begin installing the Integrated Development Environment (IDE).

Eclipse is the widely used open source IDE for Java development. Spring has extended eclipse IDE and created its own tool known as Spring Tool Suite (STS). So far, you have worked on Eclipse and must be familiar with its interface, now let us also try STS as Spring supports this better. At the time of writing, STS 4 was available for download. For this learning exercise, we will use STS where the experience is very similar to eclipse IDE.

Please download and install the required STS IDE from the URL <https://spring.io/tools>.

On the above page you will see the screen shown in Figure 22.1. Select the build based on your operating system. In our case, we have downloaded the Mac version.

### QUICK CHALLENGE

List the differences between eclipse and Spring Tool Suite.

Once the download is done, find the build and install it on your system. This installation will be similar to other applications you install on your computer. Once the installation is done, you will find the icon of the STS build, which you can use to open the STS IDE (Figure 22.2).

Once you open the IDE, you will see the interface shown in Figure 22.3.

If you are new to eclipse environment, we recommend you go through the official documentation to get familiarise yourself with the environment. You can find some useful information on the official page [https://www.eclipse.org/getting\\_started](https://www.eclipse.org/getting_started).

Once you are familiar with the interface, you can start creating a Spring MVC based REST web service using Spring Boot.

The screenshot shows the Spring Tool Suite 4 interface running on Eclipse. The workspace contains a Java class named `GreetingController.java` with the following code:

```

package hello;
import java.util.concurrent.atomic.AtomicLong;
public class GreetingController {
    private static final String template = "Hello, %s!";
    private final AtomicLong counter = new AtomicLong();
    public String greeting(String name) {
        return String.format(template, "Greeting from %s", name);
    }
}

```

The interface includes standard Eclipse toolbars and menus like File, Edit, View, Tools, Window, Help, and a Quick Access search bar. A central view displays the code editor, a package explorer showing projects like `greet-rest-service-complete`, and a central workspace area.

**Spring Tools 4 for Eclipse**

The all-new Spring Tool Suite 4.  
Free. Open source.

**Download STS4**  
Linux 64-bit

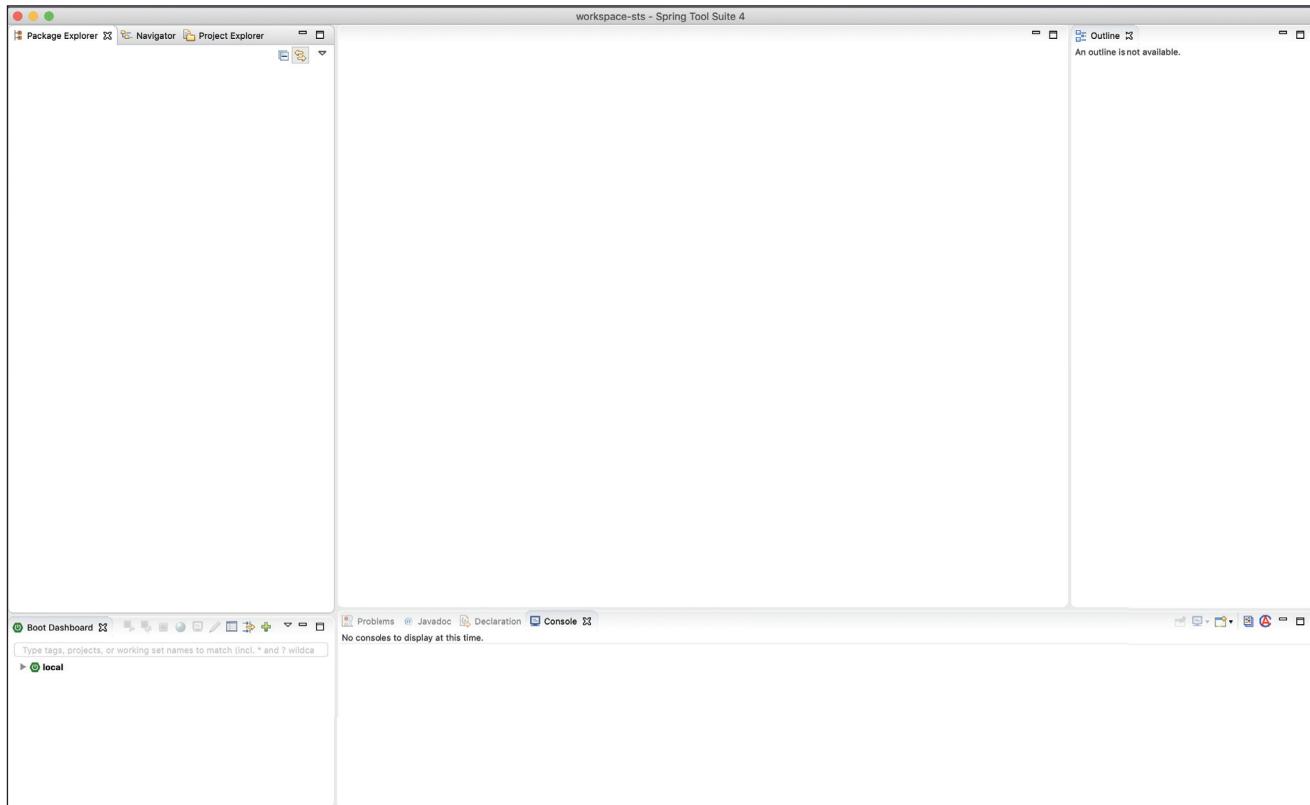
**Download STS4**  
macOS 64-bit

**Download STS4**  
Windows 64-bit

Figure 22.1 Spring Tool Suite 4 download page.



**Figure 22.2** Spring Tool Suite 4 icon.



**Figure 22.3** Spring Tool Suite 4 IDE.

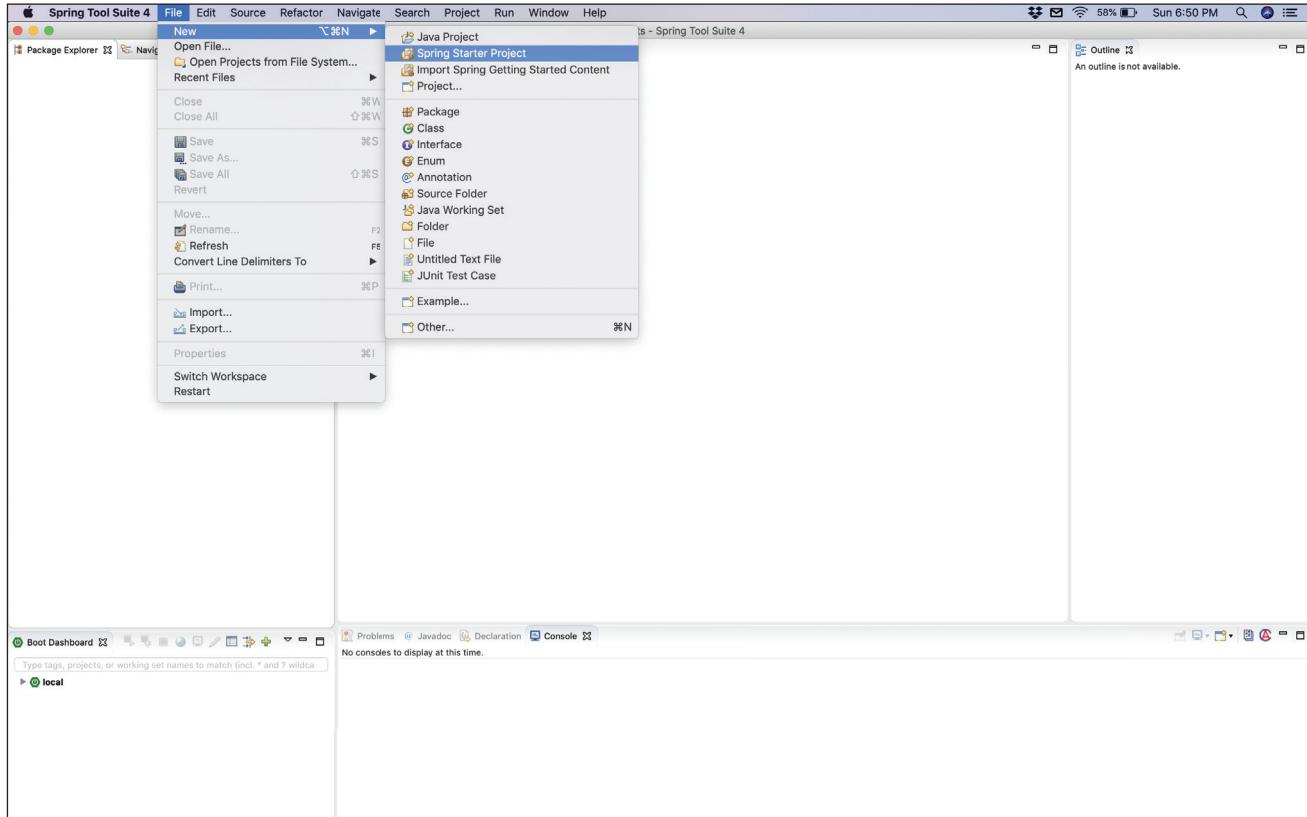


What is Spring Boot? What is the use of it?



## 22.2 | Creating a New Project

The first step is to click on the File menu. Then click on New->Spring Starter Project. See Figure 22.4 to find the location of this menu.



**Figure 22.4** Menu to create a new project.

Once you clicked on the menu, you will see the screen shown in Figure 22.5, where you will enter the project related information as follows:

1. **Service URL:** This field gives us the standard URL which we can keep it as it is.
2. **Name:** This field is to specify the project name. In our case, we will name our project as MyEShop.
3. **Location:** This file is to specify project location where the project will be saved.
4. **Type:** In our case, we will be selecting Maven. Maven is a build automation tool which greatly simplifies the build process. We will see how easy it is to manage the dependencies with Maven.
5. **Java version:** In this field, we will specify the Java version which we will use in the development. In our case, we will select Java 11.
6. **Packaging:** This field is for selecting the packaging type we need for our project. We are developing web services that will be consumed by our front end; hence we can select War as a packaging type.
7. **Language:** This option is to specify our programming language. In our case, it is Java.
8. **Group:** This field is to specify the group for our project. We can specify any text we like.
9. **Artifact:** This will be the name of the project we will be creating.
10. **Version:** This field is to specify the project version. Since we are creating a new project, we can start with 0.0.1-SNAPSHOT version. We can start with any number we like.
11. **Description:** This field is to give the project description. It is a good practice to give good project description.
12. **Package:** This field is to specify the primary package for our project.
13. **Working set:** This section is to add our project to an existing working set. It is an optional step so we can skip it.

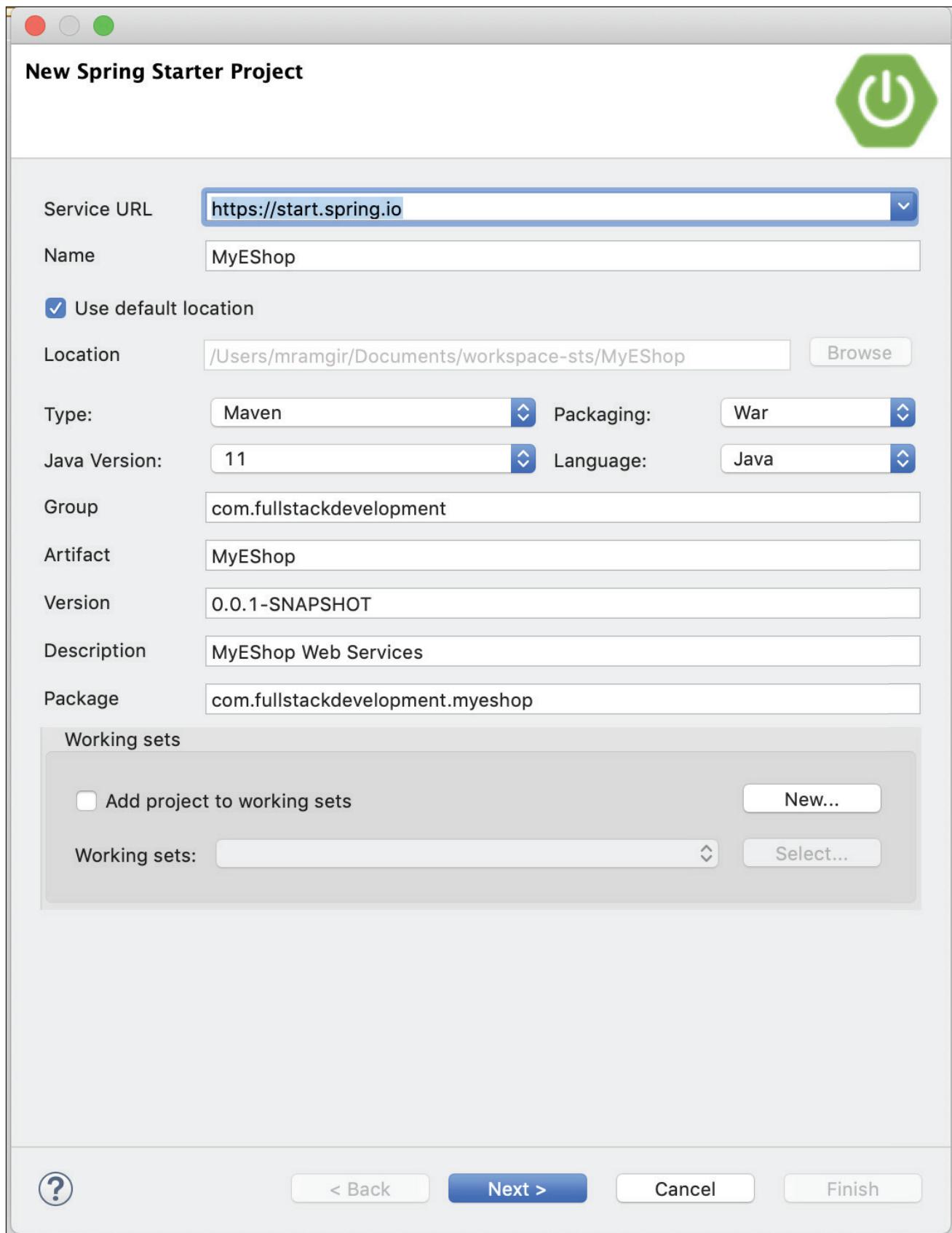


Figure 22.5 New starter project details window.

After you click on the Next button, you will see the next screen (Figure 22.6), which allows you to select project dependencies. You will see a lot of options that you can add to your project. In our case, we will be very selective in choosing, based on our requirement. In this screen, you will see an option called *Spring Boot Version*. Select the latest one you can see in the dropdown.



**Figure 22.6** New starter project dependencies window.

For our needs, we will select web package which will allow us to use Spring MVC and a few SQL options, such as JPA and MySQL, that will allow us to connect with database using object-relational mapping (ORM) tools like Hibernate (Figures 22.7 and 22.8).

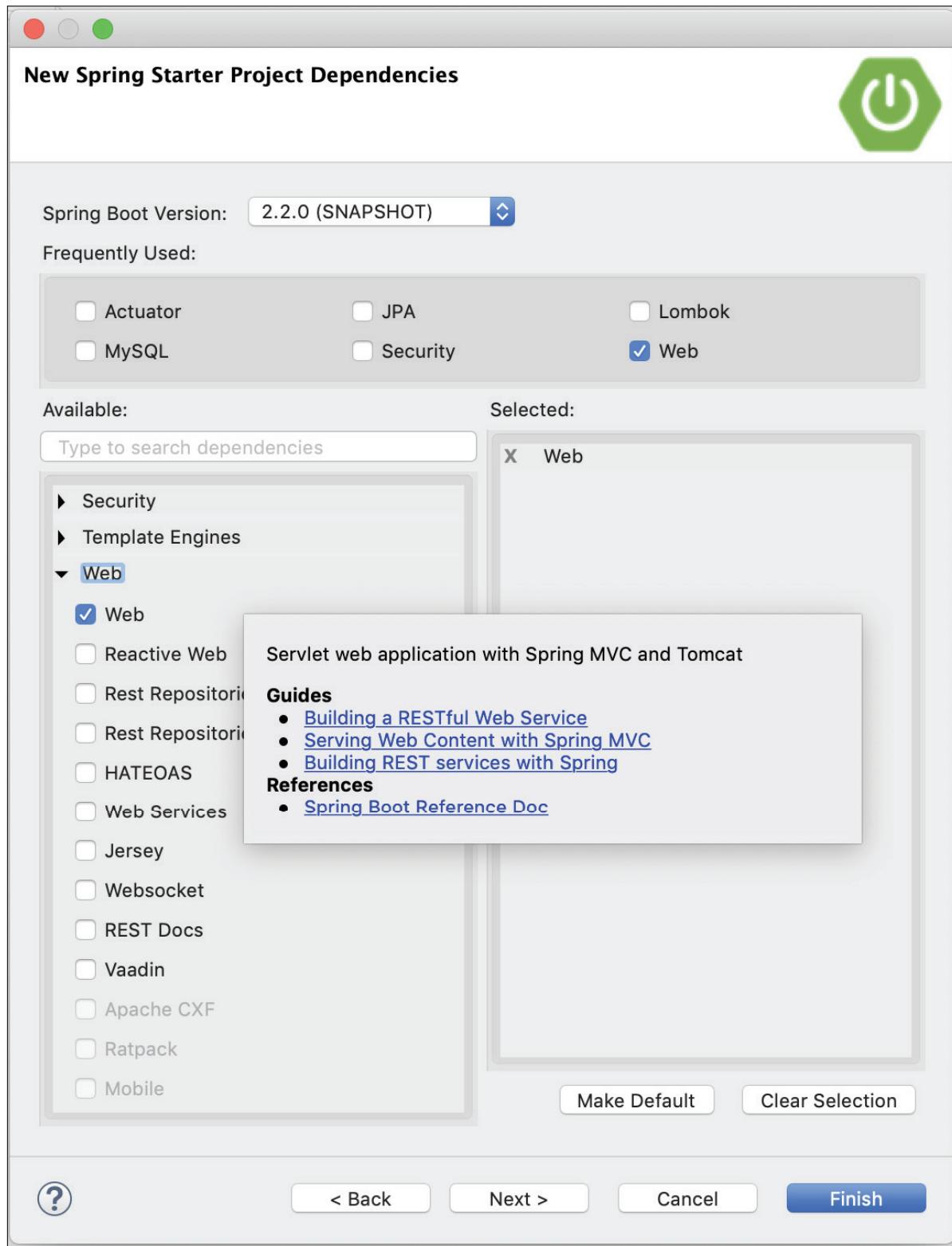


Figure 22.7 Select web package option.

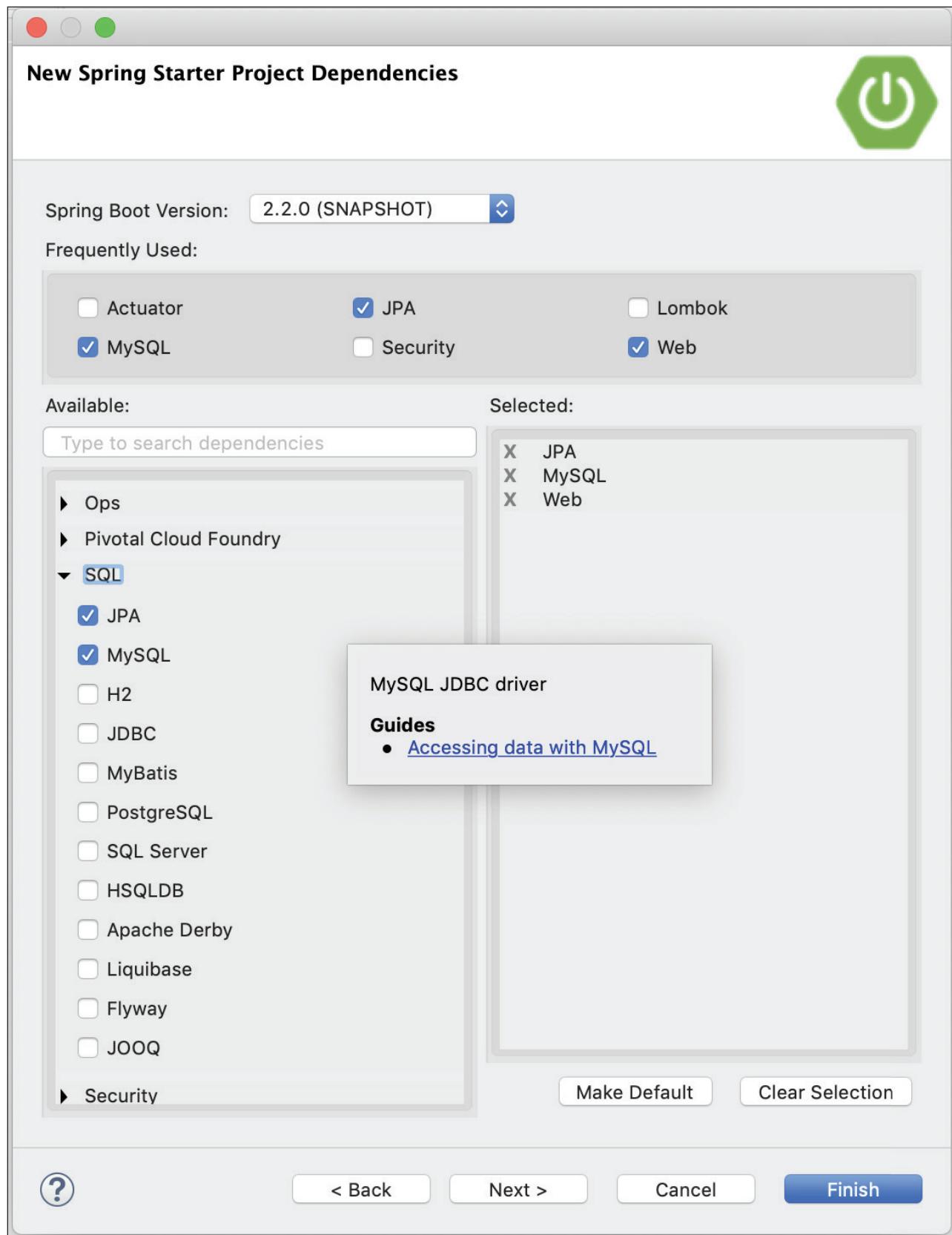
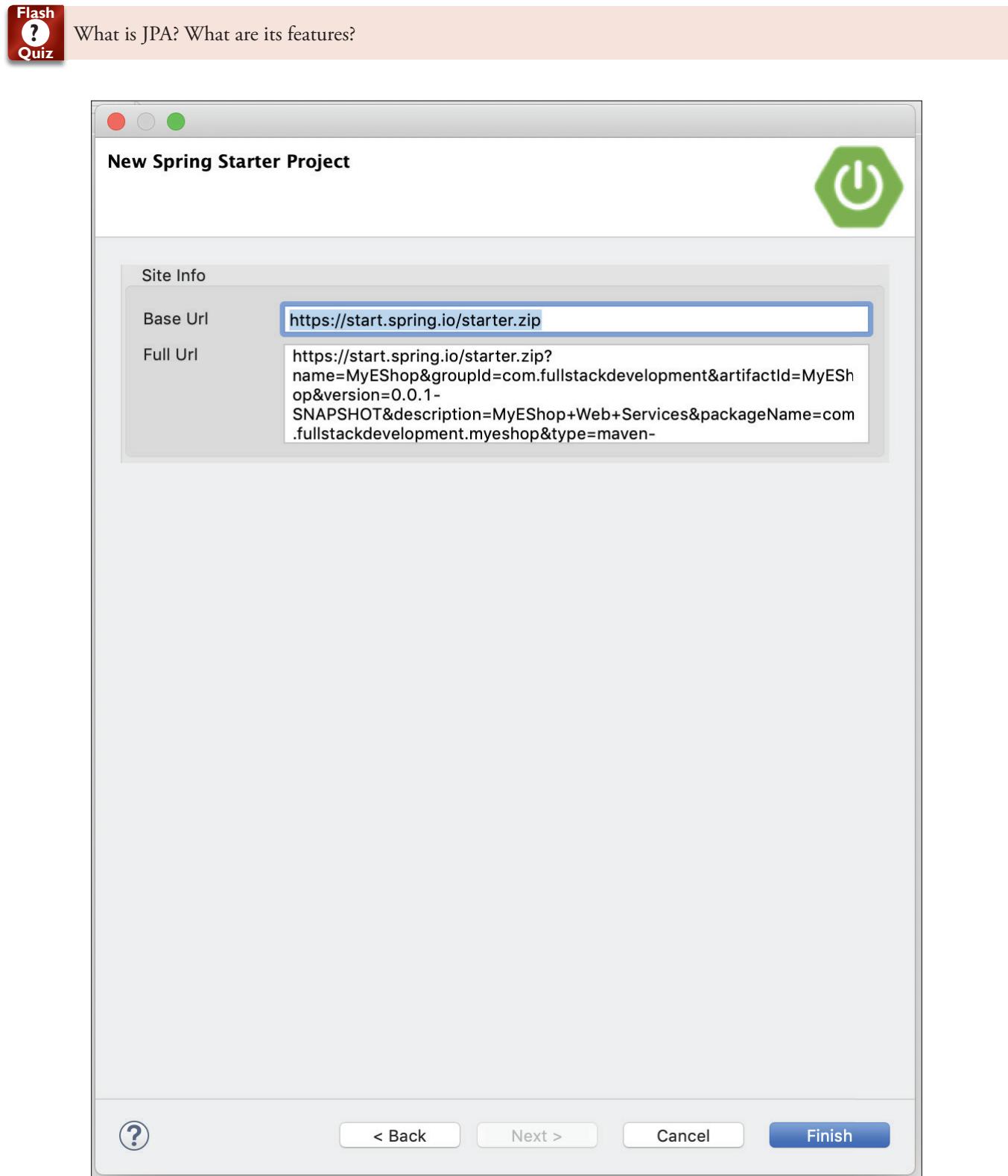


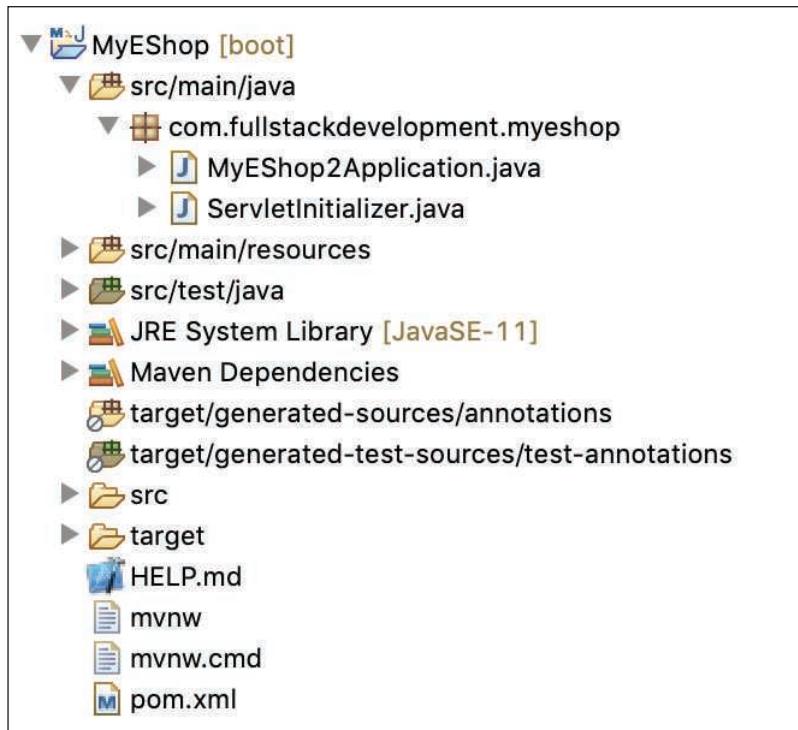
Figure 22.8 Select JPA and MySQL options.

After the desired selection, you may click on the **Next** button to continue. Upon clicking on the **Next** button, you will see the screen shown in Figure 22.9.



**Figure 22.9** Window to show base and full URLs of the application.

There is nothing we need to do on this screen, as Spring Boot will create the project we need. So just click on the **Finish** button, which will create the project based on the values we have provided earlier.



**Figure 22.10** Explorer view of the project to see all the files.

Figure 22.10 shows the project structure. The blue icon shows the project name MyEShop. After that, we have initial files that Spring Boot has created for us under src/main/java. Under this folder we have our package folders like com/fullstackdevelopment/myeshop, which is shown in STS as com.fullstackdevelopment.myeshop. At the very end, you can see a file called pom.xml. This file is a Maven file, which contains the dependency settings.

Let us explore pom.xml file and learn about specifying dependencies for our project. The first section that is shown below is for specifying the project details.

```

<groupId>com.fullstackdevelopment</groupId>
<artifactId>MyEShop</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>
<name>MyEShop</name>
<description>MyEShop Web Services</description>
  
```

These are the details we have provided while creating the project through New->Spring Starter Project menu.

**QUICK  
CHALLENGE**

Explain pom.xml in detail. List at least two of its benefits.

The next section given below is for specifying the project dependencies.

```
<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-web</artifactId>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

In the above image, you can see three dependencies that we have selected while creating the project and the fourth one is added by Spring Boot. The following shows the repositories information, which are part of our project.

```
<repositories>
    <repository>
        <id>spring-snapshots</id>
        <name>Spring Snapshots</name>
        <url>https://repo.spring.io/snapshot</url>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
    </repository>
</repositories>
```

With this we are done setting up our project environment. Now, let us begin with creating required web services using Spring MVC.



What is dependency in pom.xml?

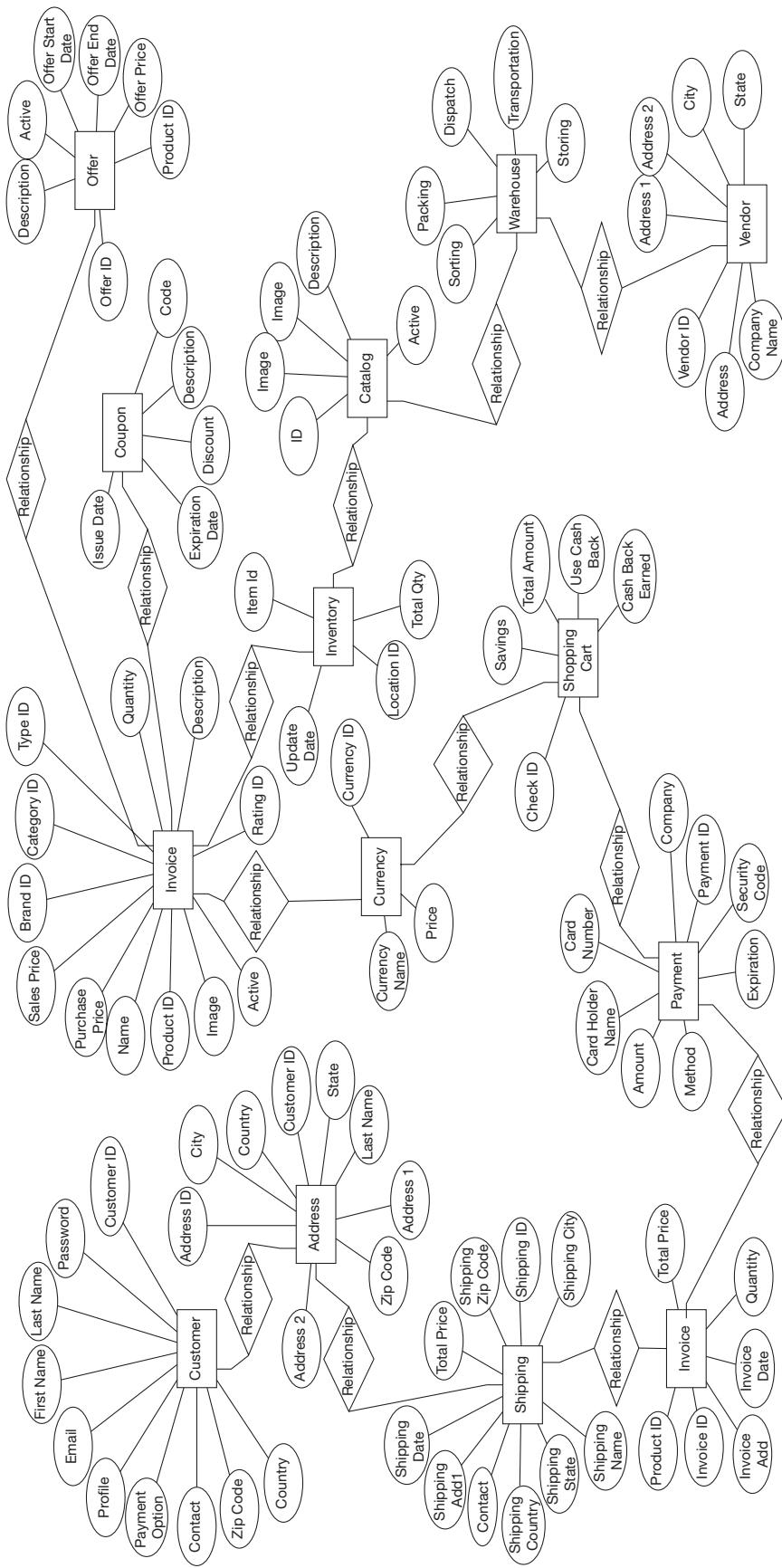


Figure 22.11 Entity relationship diagram of e-commerce entities.

## 22.3 | Creating Models

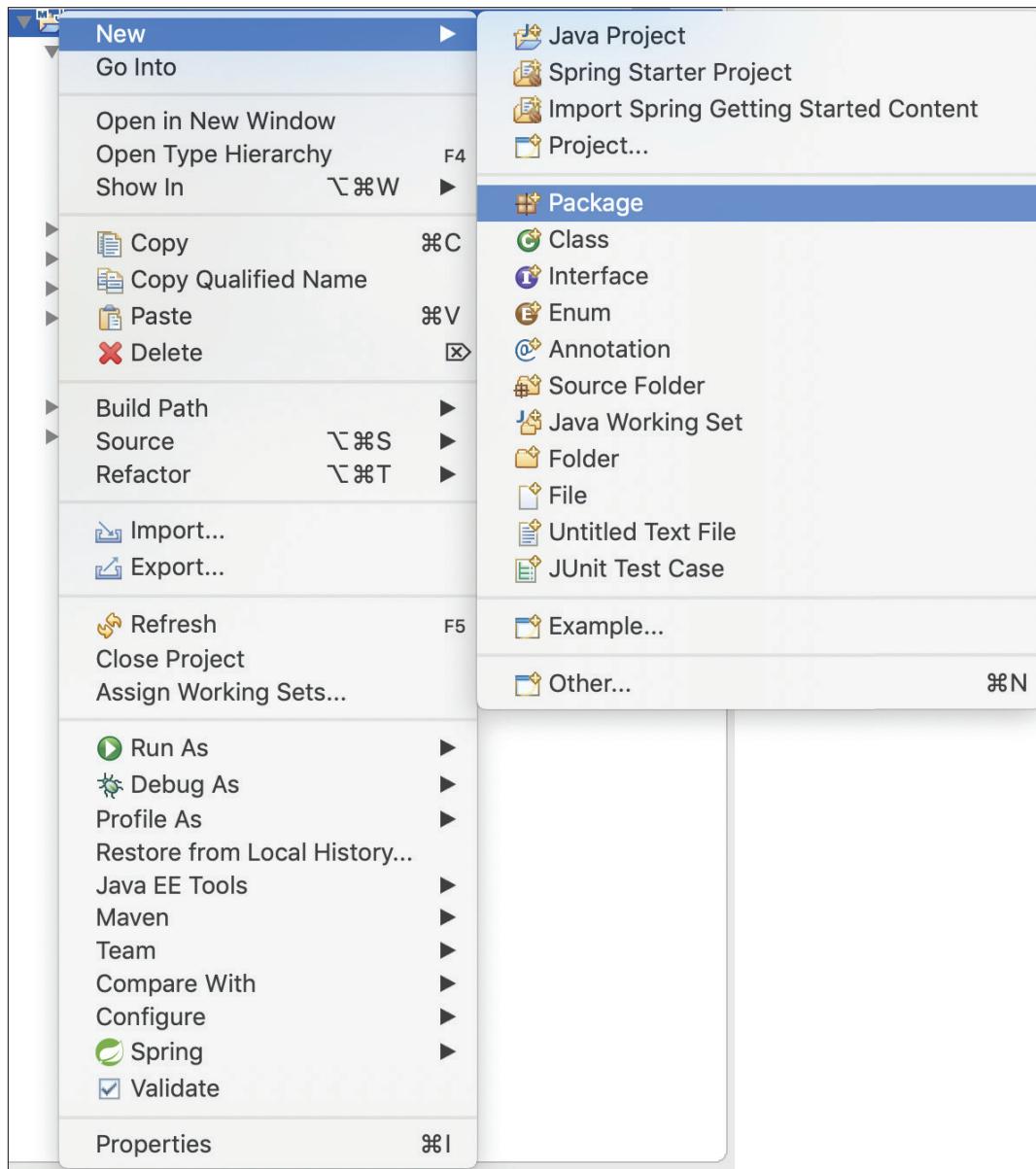


In this section, we will create models which are POJOs but are correlated with database tables by Hibernate. We will see this connection in Chapter 23, when we will explore Hibernate examples. But for now, let us look at these POJOs from database tables' point of view. We will use the Entity Relationship Diagram (ERD), which we have created earlier and that you have learned in Chapter 2. This ERD will give us an idea about the models we need to create. These are the entities defined in the ERD. Let us see the diagram again to refresh your memory (Figure 22.11).



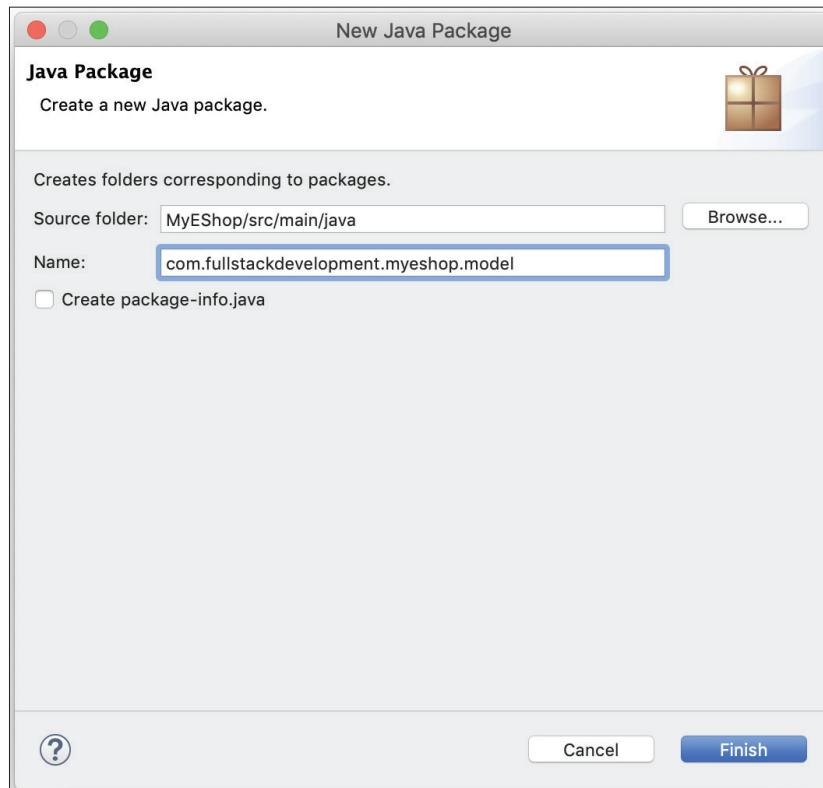
Which of the two is considered as an entity – persistence classes as entities or non-persistence classes? Explain

Now, let us create a package for Models. Right click on the project name and click on New->Package as shown in Figure 22.12.



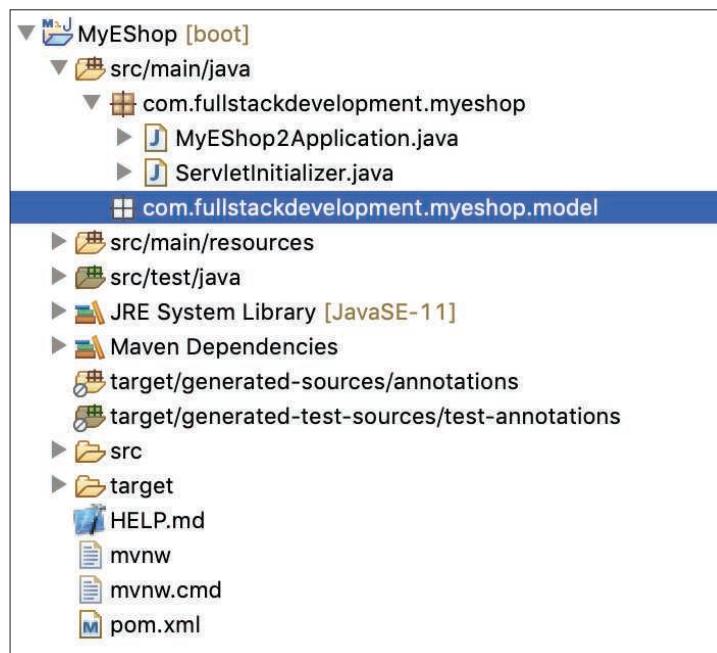
**Figure 22.12** Menu to create a package.

After clicking on the menu, you will see the screen shown in Figure 12.13, where you can specify the name of the package.



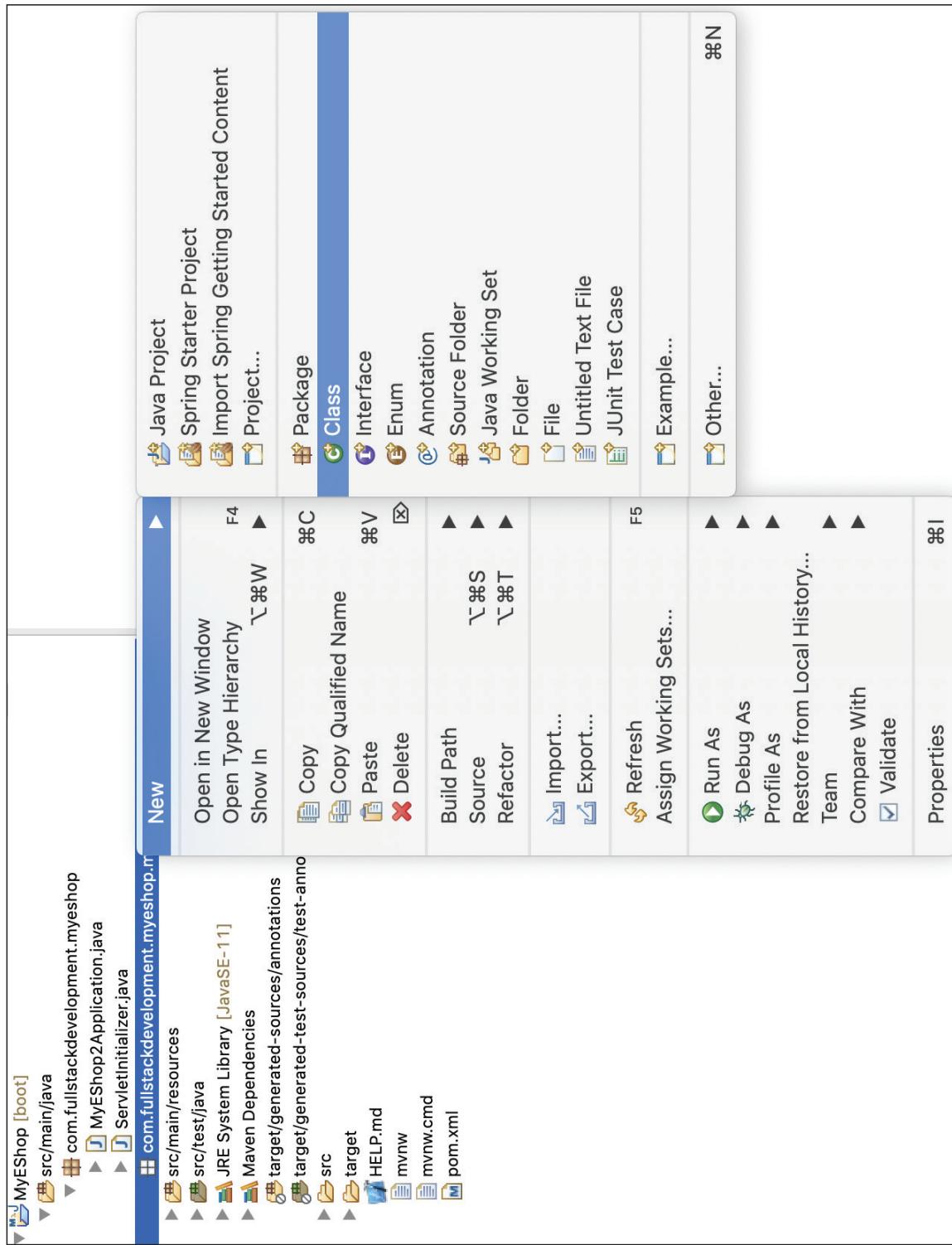
**Figure 22.13** Create Java package window.

Once you click on the **Finish** button, you will see the newly created package in the package explorer as shown in Figure 22.14.



**Figure 22.14** Newly created Java package.

Now, let us add model POJOs that we need. For this, right click on the newly created package `com.fullstackdevelopment.myeshop.model` and click on "Class" as shown in Figure 22.15.



**Figure 22.15** Menu to create a new class.

Once you click on the menu, you will see a window in which you can specify the class name as shown in Figure 22.16. In this case, we will be creating Customer model.

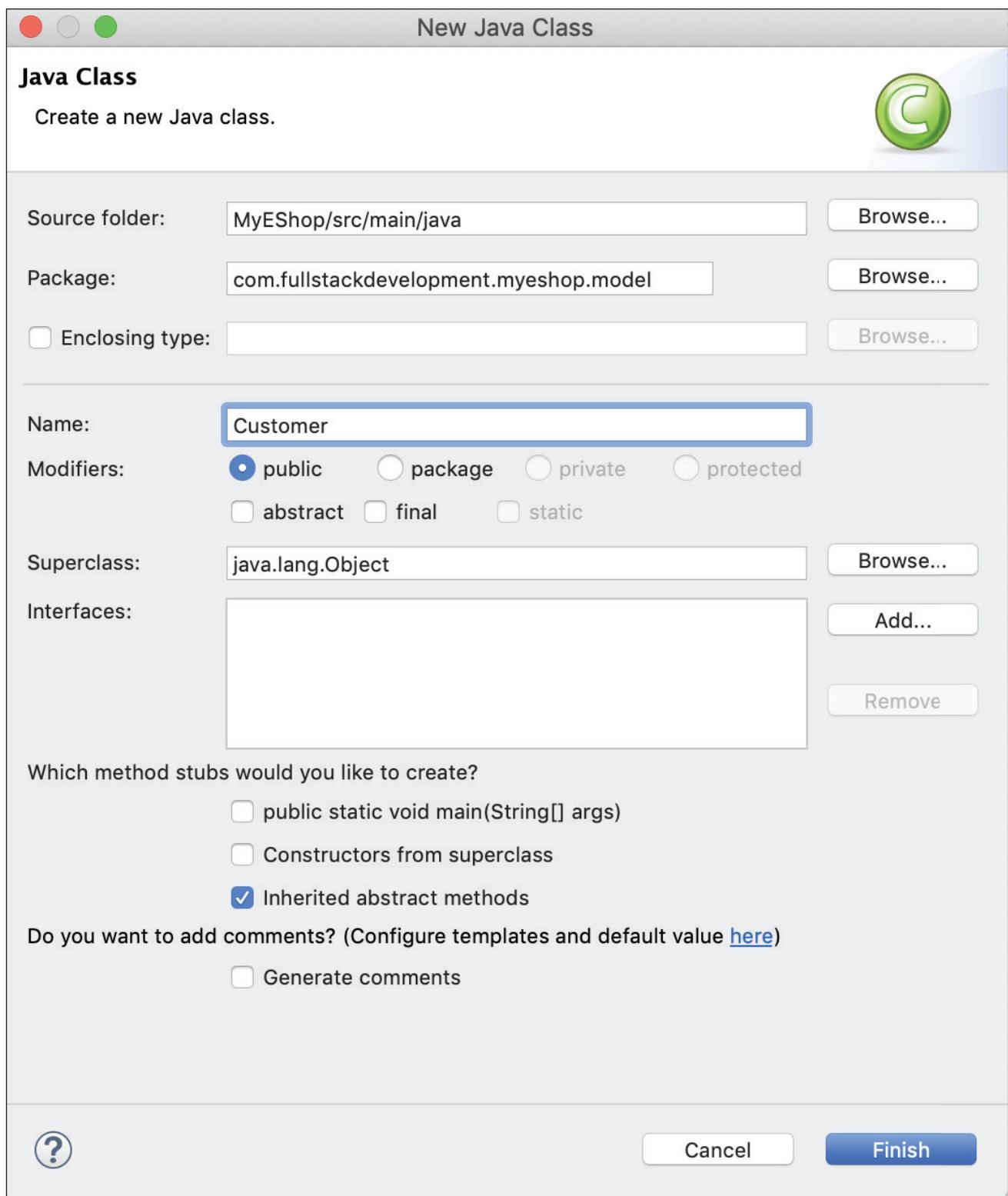
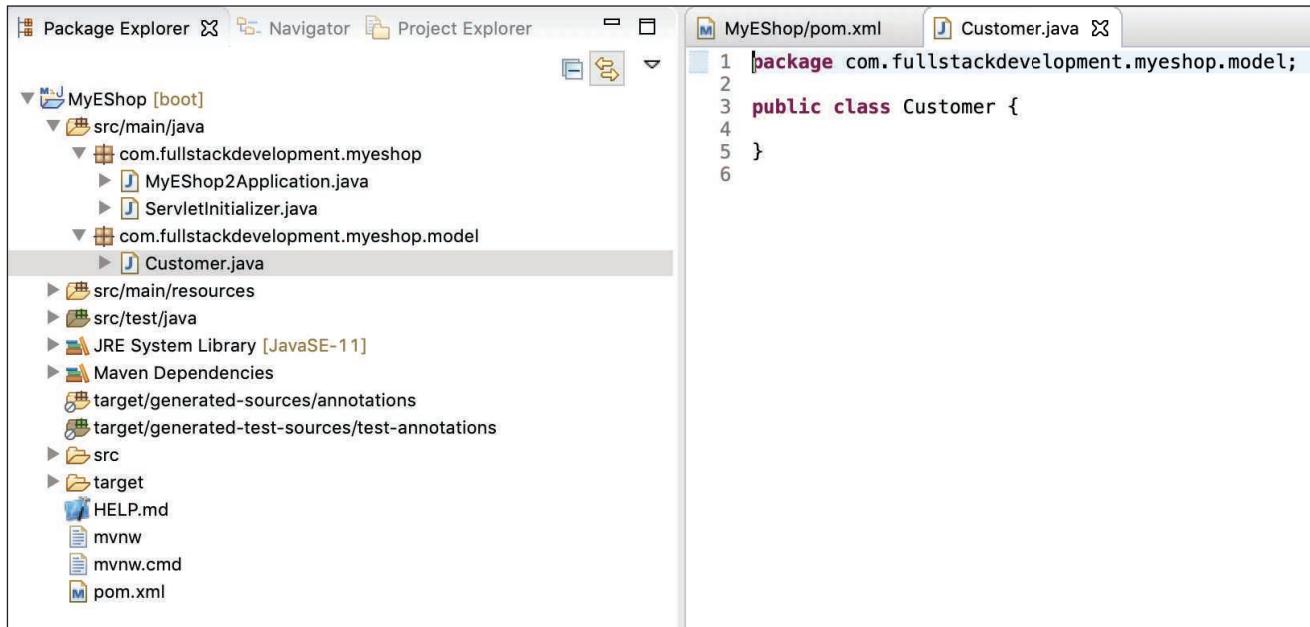


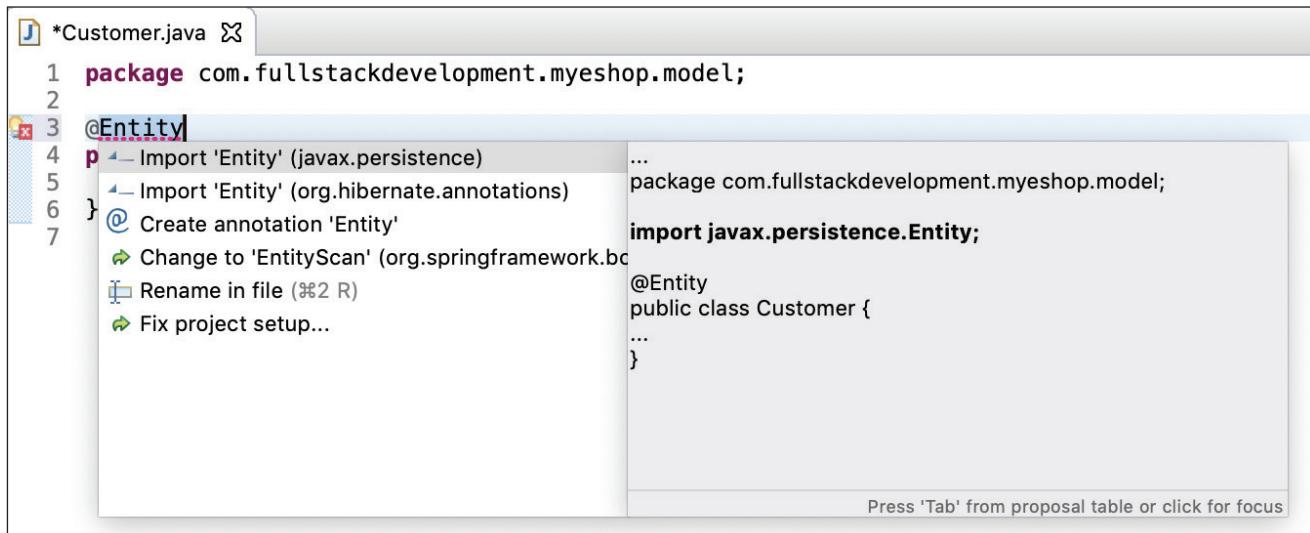
Figure 22.16 Java class details window.

Once you click on the Finish button, you will see the newly created Customer class in the package explorer under package com.fullstackdevelopment.myeshop.model. See Figure 22.17.



**Figure 22.17** Newly created class.

Now we need to define this Customer class as Entity in order to use it with Hibernate. It is as simple as adding @Entity annotation above the Customer class definition as shown in Figure 22.18.



**Figure 22.18** Auto-suggest dropdown for import.

We need to import the Entity annotation in order to use it. As you can see in Figure 22.18, there are two options – one from javax.persistence and other from org.hibernate.annotations. Although we will be using Hibernate, we do not want to make this very specific to Hibernate and hence we will be selecting Entity from javax.persistence package.

```

J *Customer.java ✘
1 package com.fullstackdevelopment.myeshop.model;
2
3 import javax.persistence.Entity;
4
5 @Entity
6 public class Customer {
7
8 }
9

```

We need to tell Spring the table name this Entity is going to be associate with. For this, we need to add `@Table` annotation from `javax.persistence`.

```

J *Customer.java ✘
1 package com.fullstackdevelopment.myeshop.model;
2
3 import javax.persistence.Entity;
4
5 @Entity
6 @Table
7 p Import 'Table' (javax.persistence)
8   ↗ Import 'Table' (org.hibernate.annotations)
9 } @ Create annotation 'Table'
10   ↗ Change to 'Tables' (org.hibernate.annotations)
    ↗ Rename in file (⌘2 R)
    ↗ Fix project setup...
... import javax.persistence.Entity;
import javax.persistence.Table;
...

```

Press 'Tab' from proposal table or click for focus

Once the import is done, we need to specify the name attribute. In our case, we will name the table as “customer”.

```

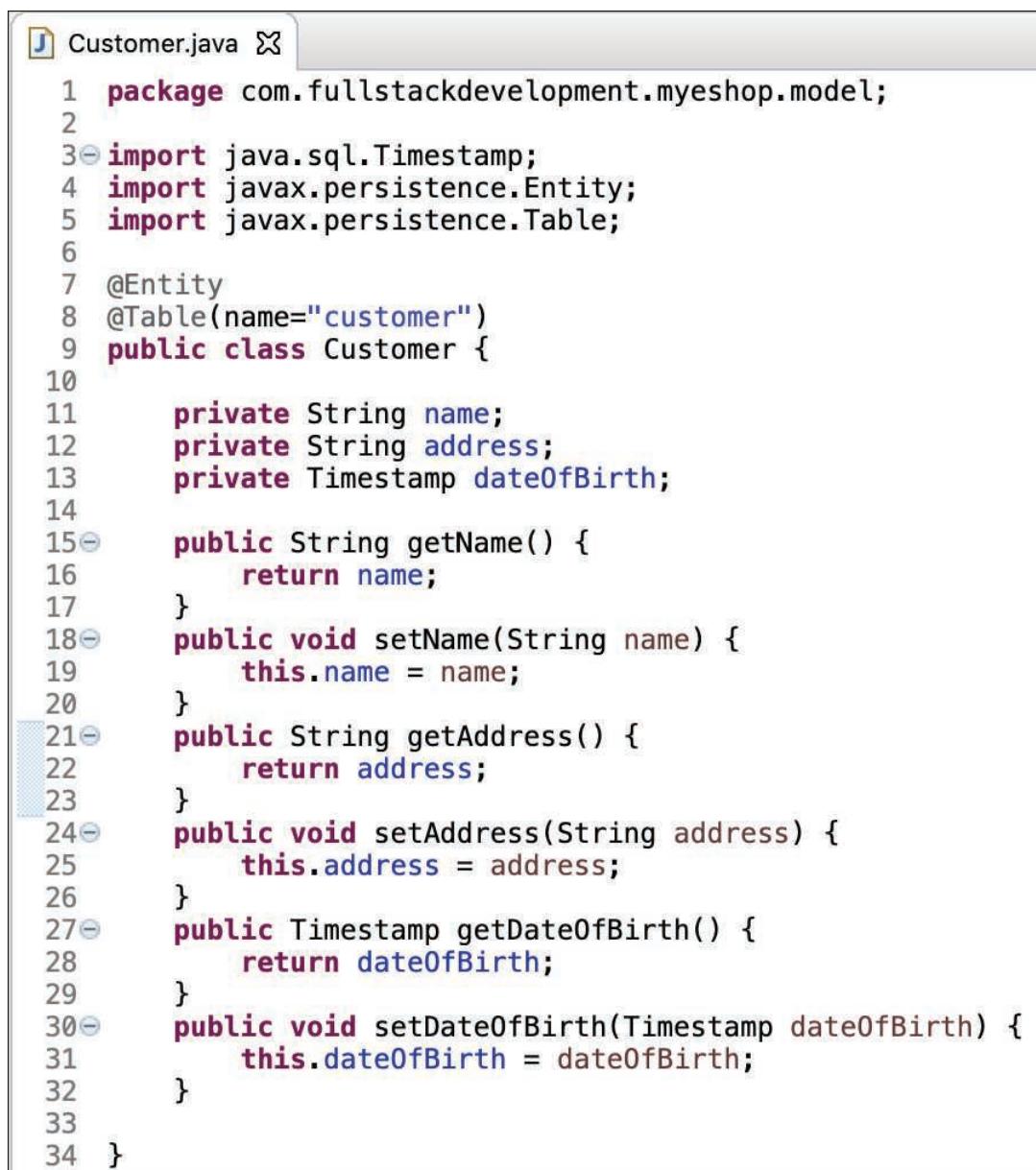
J Customer.java ✘
1 package com.fullstackdevelopment.myeshop.model;
2
3 import javax.persistence.Entity;
4 import javax.persistence.Table;
5
6 @Entity
7 @Table(name="customer")
8 public class Customer {
9
10 }
11

```

Now, it is time to add attributes to the Customer model. These attributes can be customer name, customer address, customer date of birth, etc. To start, we will add simple fields and later we can see the improved version of this entity with other entities related to it. For example, for now we have used address as a String field. However, address can be a separate model that can store multiple addresses such as shipping address, billing address, etc. So, this address entity will be related to Customer entity in “many-to-one” fashion and from Customer point of view in “one-to-many” fashion. In Chapter 23, we will explore this in detail. For now, let us continue with a simple example.

**QUICK CHALLENGE**

Define @Entity and @Table annotations in detail.



```

1 package com.fullstackdevelopment.myeshop.model;
2
3 import java.sql.Timestamp;
4 import javax.persistence.Entity;
5 import javax.persistence.Table;
6
7 @Entity
8 @Table(name="customer")
9 public class Customer {
10
11     private String name;
12     private String address;
13     private Timestamp dateOfBirth;
14
15     public String getName() {
16         return name;
17     }
18     public void setName(String name) {
19         this.name = name;
20     }
21     public String getAddress() {
22         return address;
23     }
24     public void setAddress(String address) {
25         this.address = address;
26     }
27     public Timestamp getDateOfBirth() {
28         return dateOfBirth;
29     }
30     public void setDateOfBirth(Timestamp dateOfBirth) {
31         this.dateOfBirth = dateOfBirth;
32     }
33 }
34 }
```

As you can see in the above example, we have added getter and setter for each field. This is the most important step we need to perform as Spring is going to use these fields to inject values to this Entity object. Hence, make sure you add getter and setter

for all the fields you add. The above example also shows a new field called Timestamp, which is imported from java.sql. As we will be storing this in the database, we need to make sure it is compatible with that.

**QUICK CHALLENGE**

Using the examples shown in Section 22.3 of Customer, create all other models which we have shown in the ERD. Start with basic fields.

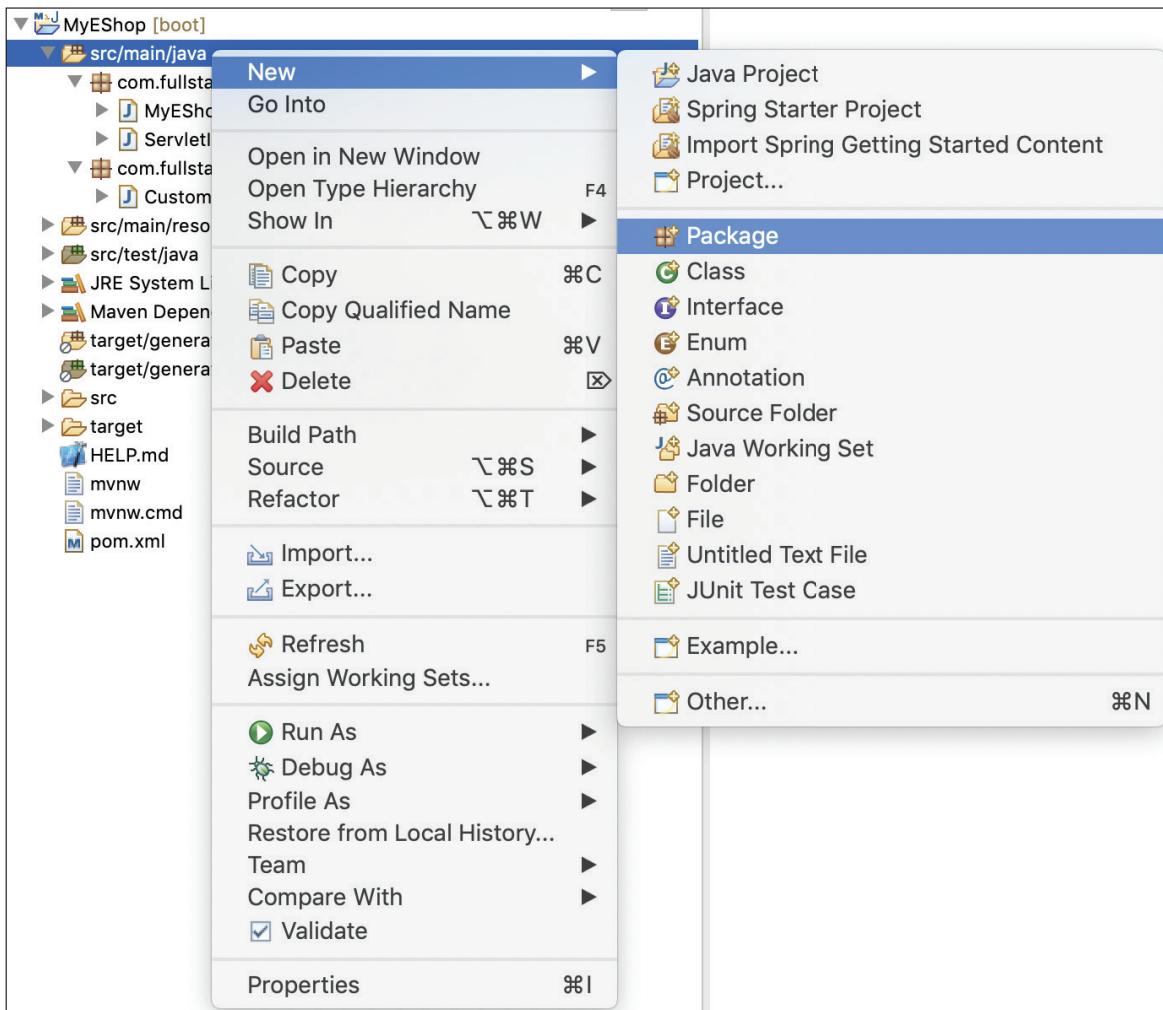
Now we will move on to creating Data Access Object (DAO) for this Customer model. This layer is important in terms of keeping database interaction related code. DAO will access models to fetch the required data.

## 22.4 | Creating Data Access Object



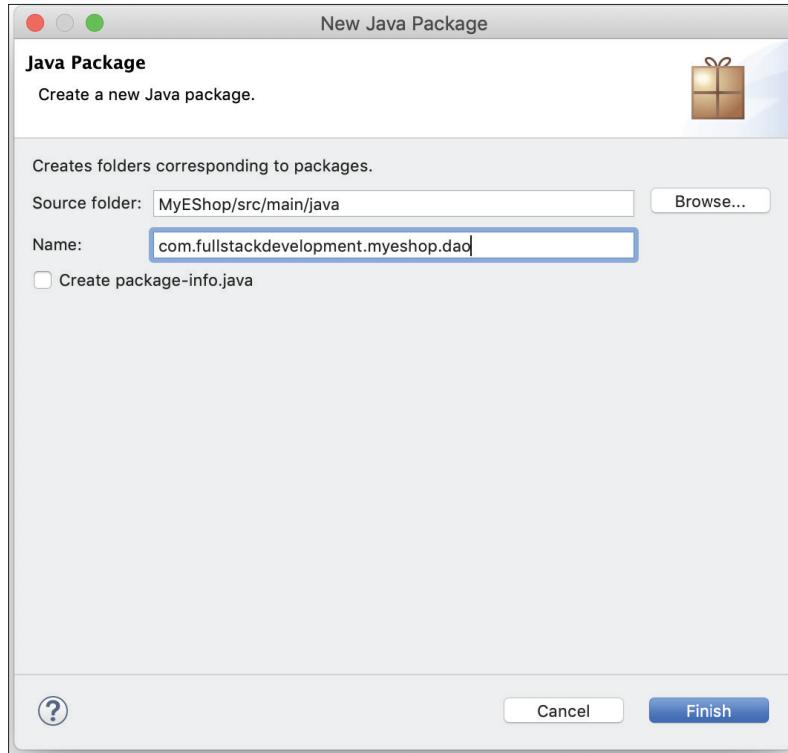
In this part, we will create DAO for the Customer model we have created in Section 22.3. DAO encapsulates the details of models (persistence layer) and provides an interface for database operations for adding, retrieving, updating, and deleting data. This mechanism supports the single responsibility principle.

Now, let us create a package for DAO. Use the same steps to create a new package as we did earlier for model. Right click on the project and click on New->Package as shown in Figure 22.19.



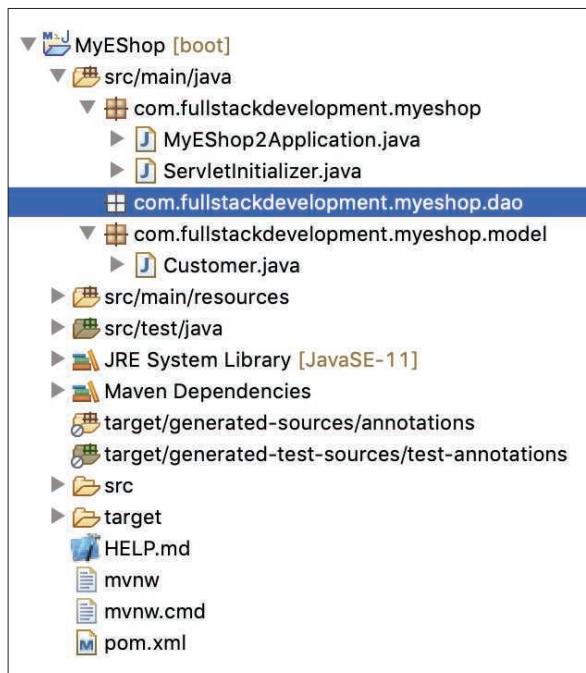
**Figure 22.19** Menu to create a new package.

Once you click on the package menu, you will see the screen to enter the details about this new package as shown in Figure 22.20.



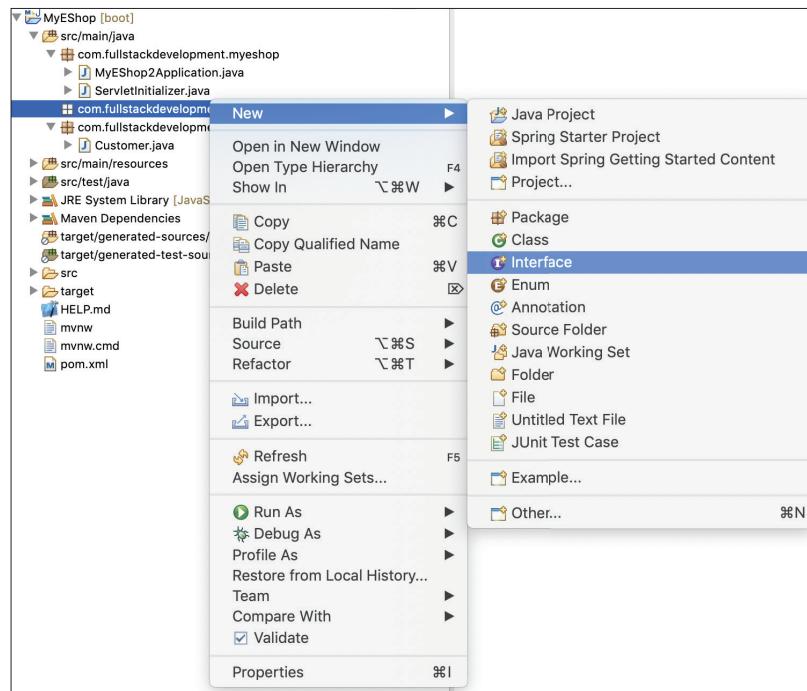
**Figure 22.20** A new package detail window.

After entering the name and clicking on the Finish button, it will show you the newly created package in the package explorer (Figure 22.21).



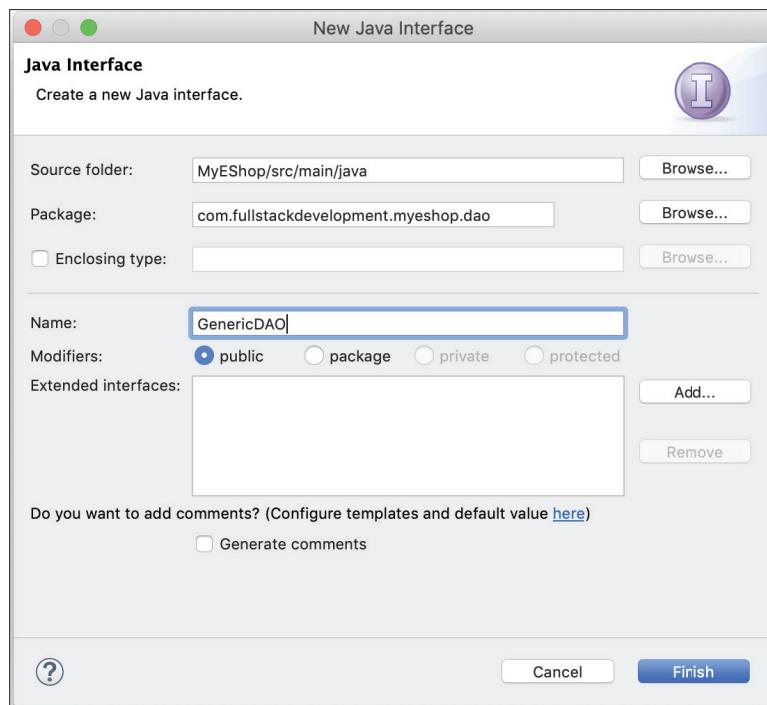
**Figure 22.21** A newly created package.

In this package, we can add our DAO. Before adding any specific DAO classes, we can create an interface which all the DAO interfaces can extend that provides operations which will be useful for all the classes like get, getAll, save, update, delete, etc. Let us create this GenericDAO interface. For this, right click on the package and click on New->Interface as shown in Figure 22.22.



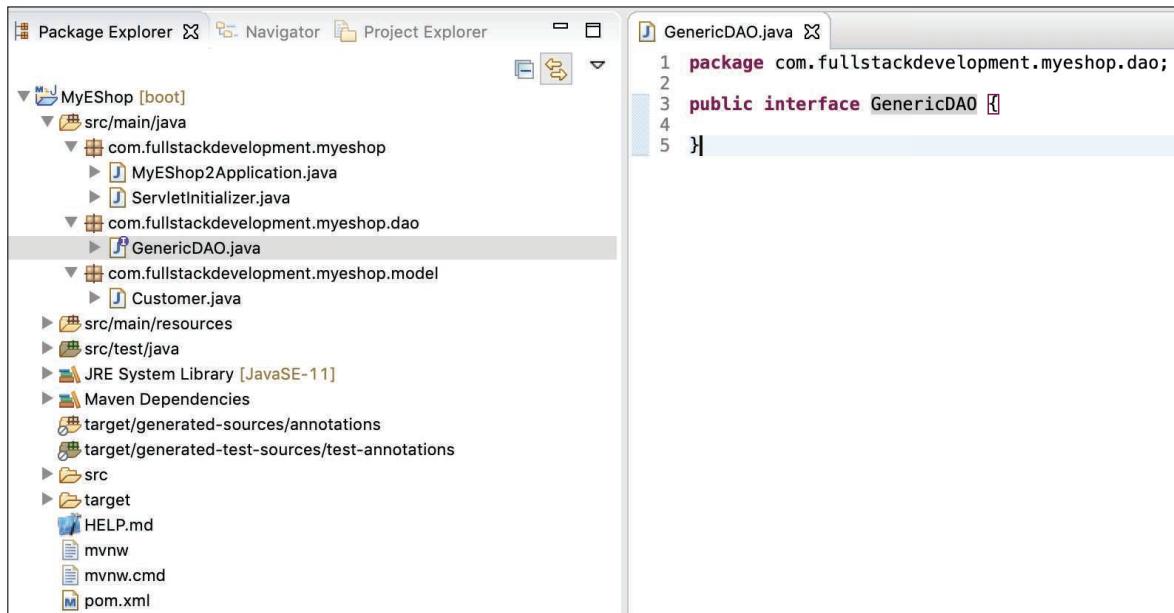
**Figure 22.22** Menu to create a new interface.

Upon clicking on New->Interface, you will see a window which will allow you to enter the interface related information. See Figure 22.23.



**Figure 22.23** New interface detail window.

Enter DAO interface name as “GenericDAO” and click on the Finish button. As shown in Figure 22.24, this will create the interface we need.

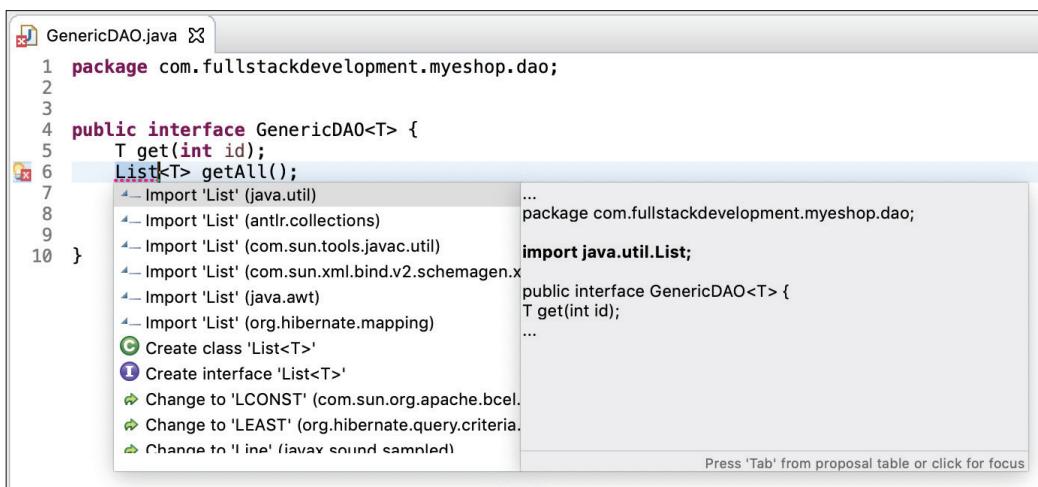


**Figure 22.24** Newly created interface.

We need to add type for this interface so the model we will be implementing in can replace it.



Now, we can add the method declarations that will be useful to all the models. It is time to import the class, take the mouse pointer to the highlighted word and click on the import package. For List, we will use java.util.



**Figure 22.25** Auto-suggest dropdown to import List class.

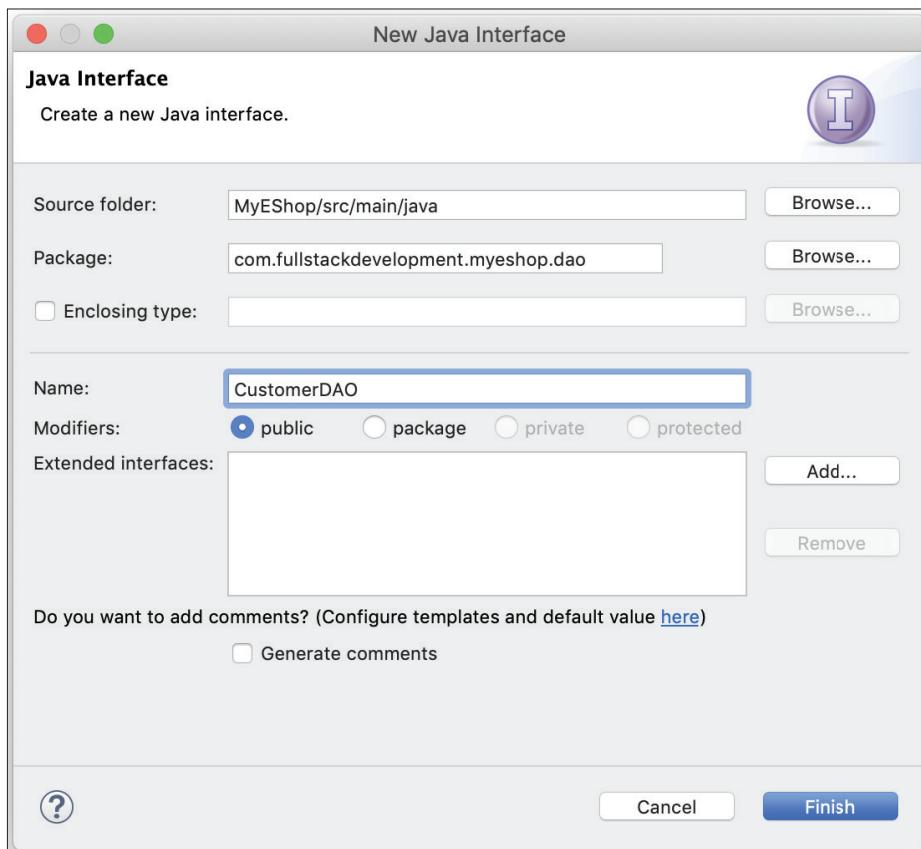
As shown in Figure 22.25, we are importing the List class from java.util package. Once the import is done, your code should look as shown below.

```

1 package com.fullstackdevelopment.myeshop.dao;
2
3 import java.util.List;
4
5 public interface GenericDAO<T> {
6     T get(int id);
7     List<T> getAll();
8     Long save(T t);
9     void update(T t);
10    void delete(T t);
11 }

```

We can extend this GenericDAO to all the DAO interfaces we will be creating. This will avoid the code duplication and we can still add model specific interface methods. Let us try and understand this better with the help of an example. So let us create a CustomerDAO interface to have Customer specific methods (Figure 22.26). Let us use the same step to create an interface and give name it CustomerDAO.



**Figure 22.26** New interface detail window.

Once you click on the Finish button, you will see the newly created interface in the project explorer under com.fullstackdevelopment.myeshop.dao package. Now, let us extend this interface from GenericDAO and use type as Customer. See the below example.

```

1 package com.fullstackdevelopment.myeshop.dao;
2
3 import com.fullstackdevelopment.myeshop.model.Customer;
4
5 public interface CustomerDAO extends GenericDAO<Customer>{
6
7 }
8

```

This is a great example of object-oriented programming (OOP) principle. In this example, we are accessing all the method declarations from GenericDAO in CustomerDAO interface and making them available to all the classes, which will implement CustomerDAO interface. In CustomerDAO interface, we are free to declare any Customer specific methods which will only be available to classes which implements CustomerDAO. See the following example.

```

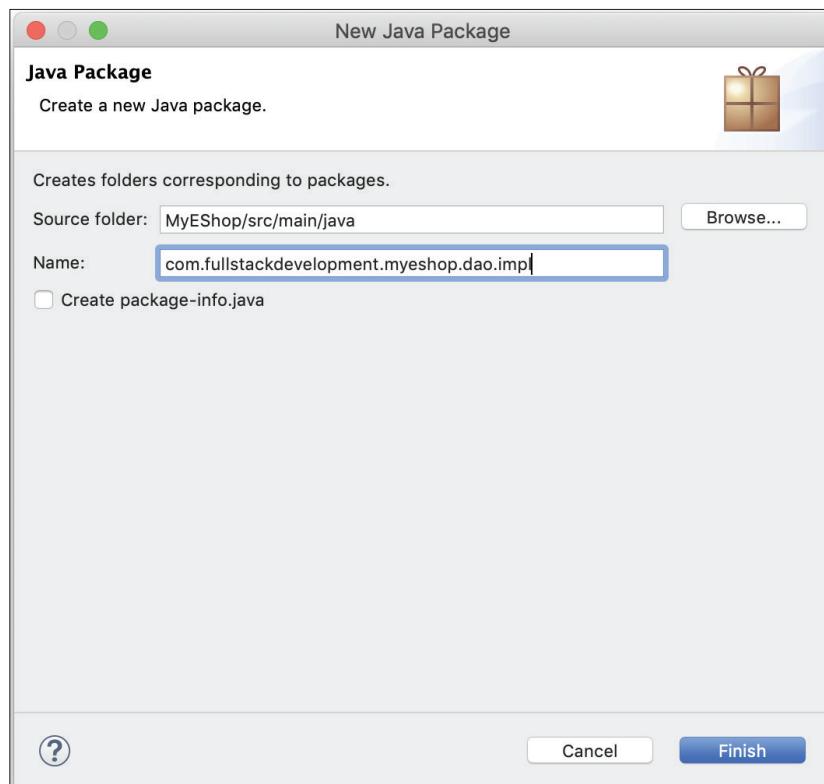
1 package com.fullstackdevelopment.myeshop.dao;
2
3 import com.fullstackdevelopment.myeshop.model.Customer;
4
5 public interface CustomerDAO extends GenericDAO<Customer>{
6     Customer findCustomerByEmail(String email);
7 }
8

```

**QUICK CHALLENGE**

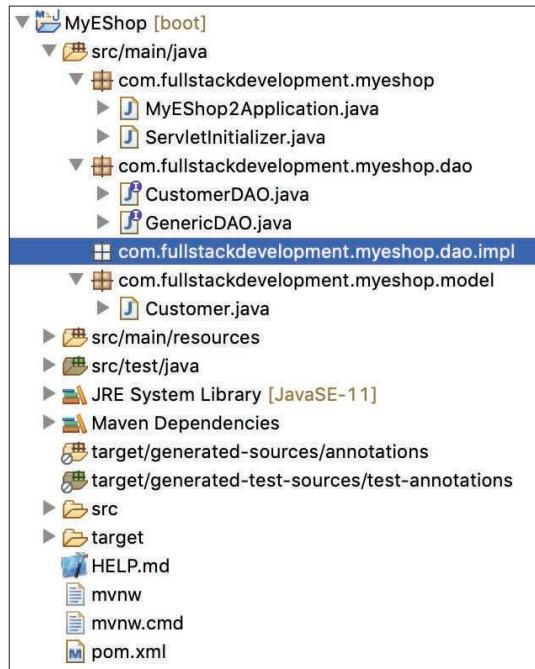
As per the examples shown in 22.4, create DAO interfaces for all the models you have created in the earlier exercise.

Now, it is time to create a concrete DAO class that will implement this CustomerDAO interface. Right click on the project and add a new package as we have added before and give name it com.fullstackdevelopment.myeshop.dao.impl as shown in Figure 22.27.



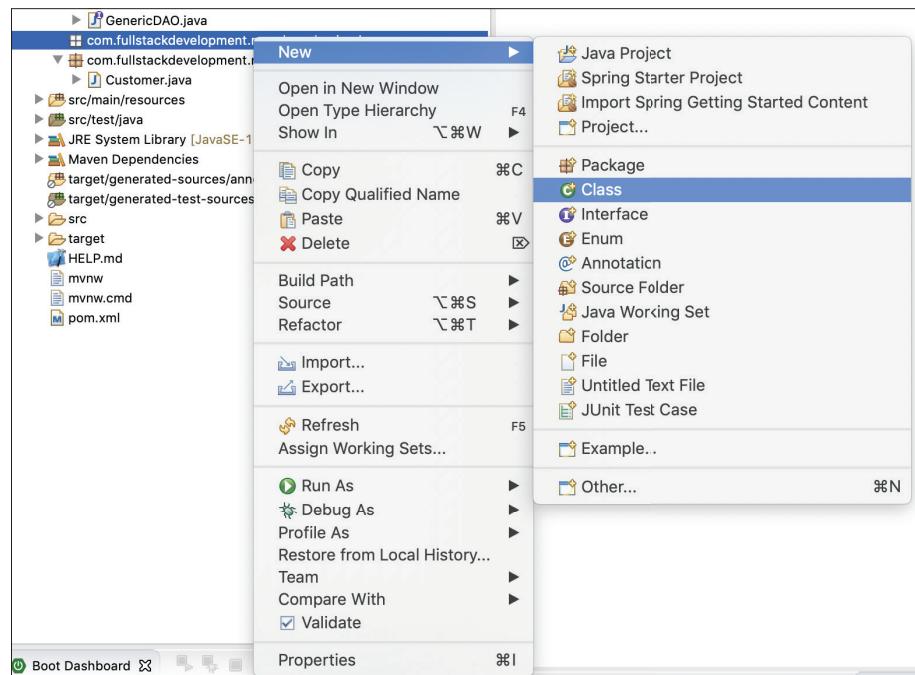
**Figure 22.27** New package detail window.

Once you click on the Finish button, you will see “com.fullstackdevelopment.myeshop.dao.impl” in the project explorer as shown in Figure 22.28.



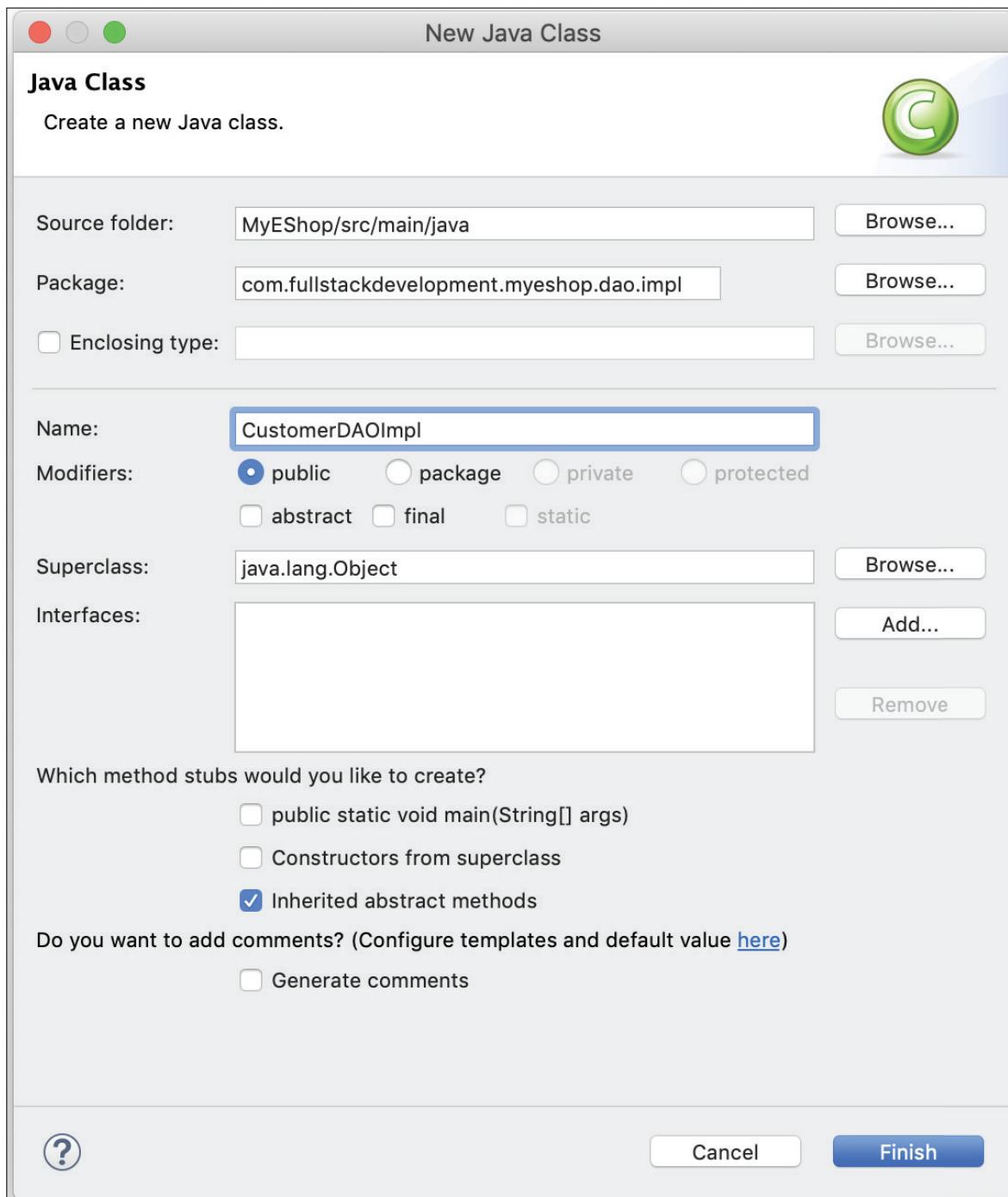
**Figure 22.28** Newly package created.

We will add all the DAO implementation classes in this package. This is just to separate the implementation classes and interfaces for code readability purpose. Now, let us add our CustomerDAOImpl class by right clicking on this new package and selecting New->Class as shown in Figure 22.29.



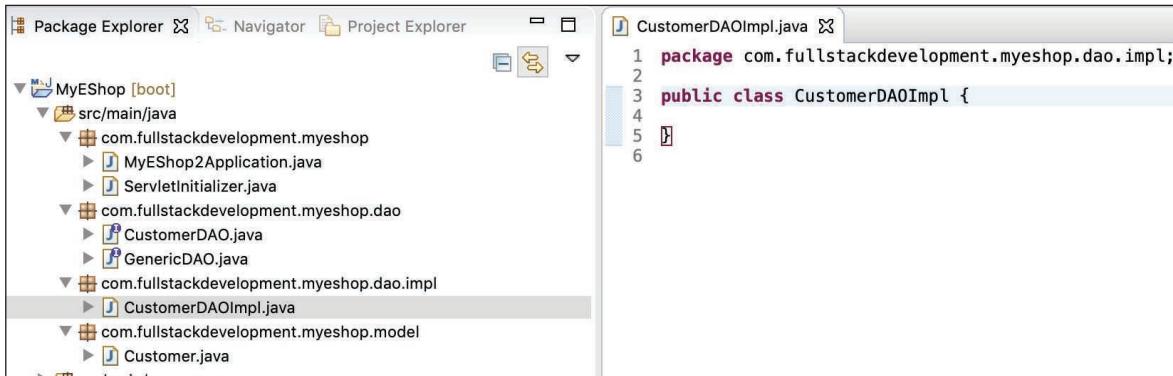
**Figure 22.29** Menu to create a new class.

Once you click on the New->Class menu, you will be presented with a window to provide the information about the class as shown in Figure 22.30.



**Figure 22.30** New class detail window.

In the name field type “CustomerDAOImpl” and click on the Finish button. This will add CustomerDAOImpl class under “com.fullstackdevelopment.myeshop.dao.impl” as shown in Figure 22.31.



**Figure 22.31** Newly created class.

Now, it is time to implement the CustomerDAO interface that we have created earlier. This interface will require us to provide a concrete implementation of the methods that are declared in the interface.

The screenshot shows the code editor for 'CustomerDAOImpl.java'. The code now includes the 'implements' keyword:

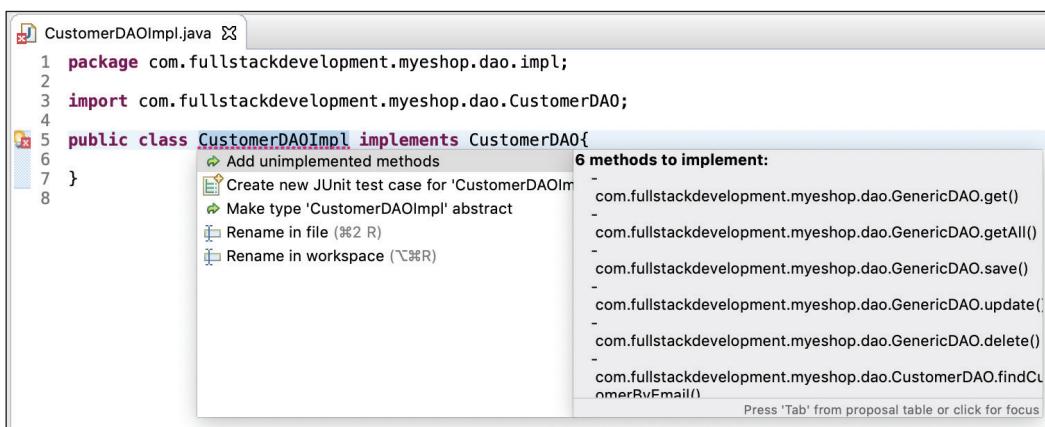
```

1 package com.fullstackdevelopment.myeshop.dao.impl;
2
3 import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
4
5 public class CustomerDAOImpl implements CustomerDAO{
6
7 }
8

```

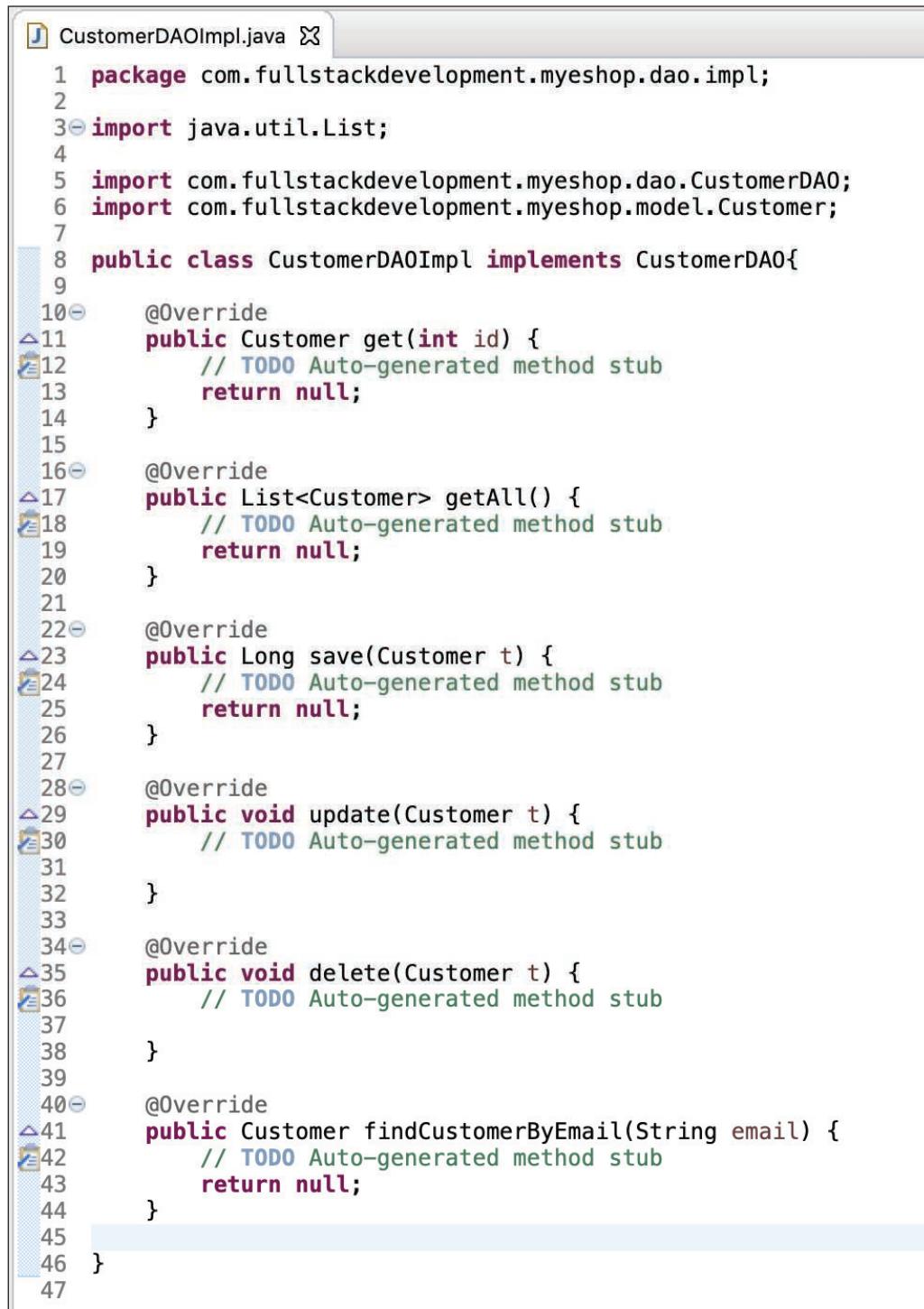
**Figure 22.32** Code to show newly created class.

The red underline on the class name 'CustomerDAOImpl' in Figure 22.32 is because of the unimplemented methods that are declared in the CustomerDAO interface. IDE makes it easier for us by complaining about it by the moment we add implements keyword with the interface name (Figure 22.33). If you take the mouse pointer to the 'CustomerDAOImpl' word, you will see the details.



**Figure 22.33** Auto-suggest dropdown to add unimplemented methods.

You can simply click on the “Add unimplemented methods” to add all the required methods’ skeleton code. Later we can add the actual code.



```

1 package com.fullstackdevelopment.myeshop.dao.impl;
2
3 import java.util.List;
4
5 import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
6 import com.fullstackdevelopment.myeshop.model.Customer;
7
8 public class CustomerDAOImpl implements CustomerDAO{
9
10    @Override
11    public Customer get(int id) {
12        // TODO Auto-generated method stub
13        return null;
14    }
15
16    @Override
17    public List<Customer> getAll() {
18        // TODO Auto-generated method stub
19        return null;
20    }
21
22    @Override
23    public Long save(Customer t) {
24        // TODO Auto-generated method stub
25        return null;
26    }
27
28    @Override
29    public void update(Customer t) {
30        // TODO Auto-generated method stub
31    }
32
33
34    @Override
35    public void delete(Customer t) {
36        // TODO Auto-generated method stub
37    }
38
39
40    @Override
41    public Customer findCustomerByEmail(String email) {
42        // TODO Auto-generated method stub
43        return null;
44    }
45
46}
47

```

As you can see in the code in image above, it has added all the methods that are declared in the GenericDAO interface and one method “findCustomerByEmail” from CustomerDAO interface. This shows how easy it is to use an OOP language like Java.

**QUICK  
CHALLENGE**

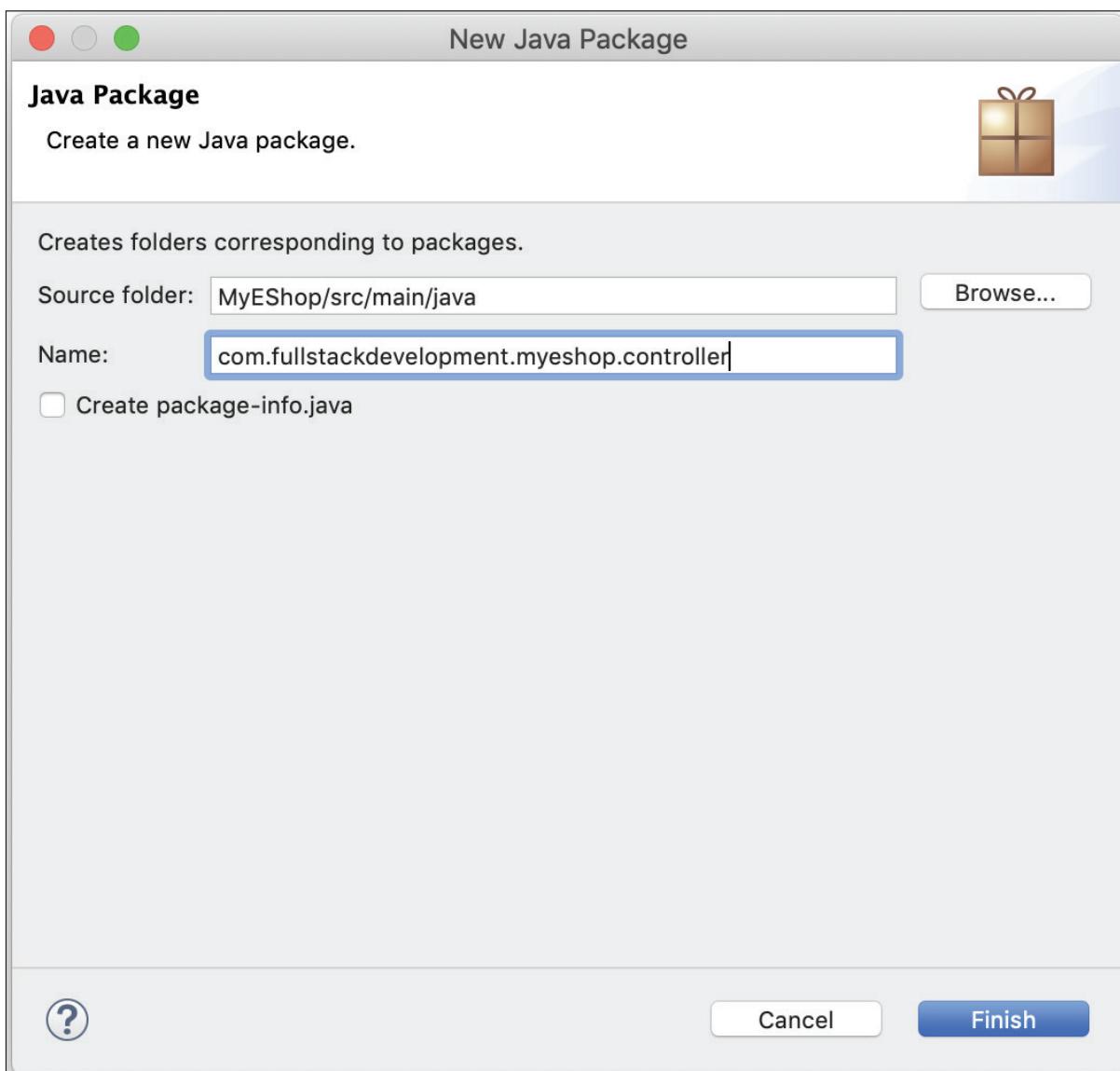
Create concrete DAO classes for the DAO interfaces you have created in the earlier Quick Challenge given in Section 22.4.

Now, we will create controller classes. We will come back and visit this DAO section in Chapter 23 when we will implement Hibernate code.

## 22.5 | Creating Controller

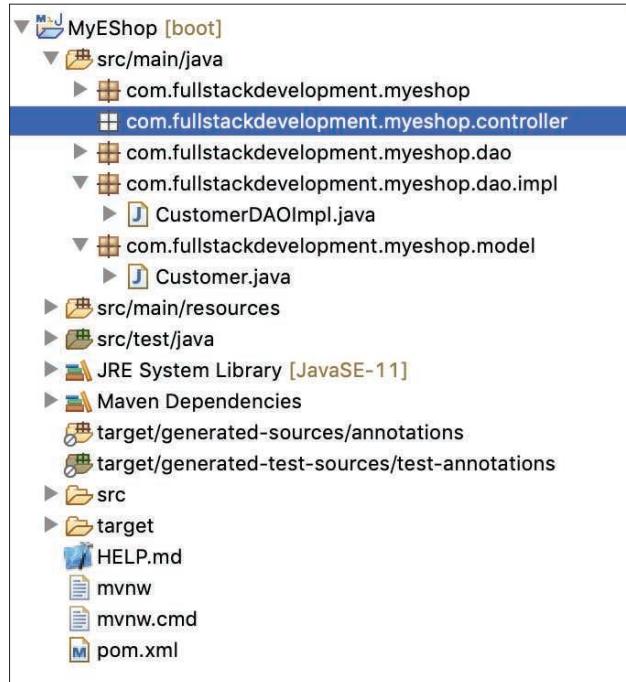


In this section, we will create controller classes for the models we have created earlier in Section 22.3. We will create a new package for these classes. So right click on the project and add a new package the way we have added earlier and give name it “com.fullstackdevelopment.myeshop.controller” as shown in Figure 22.34.



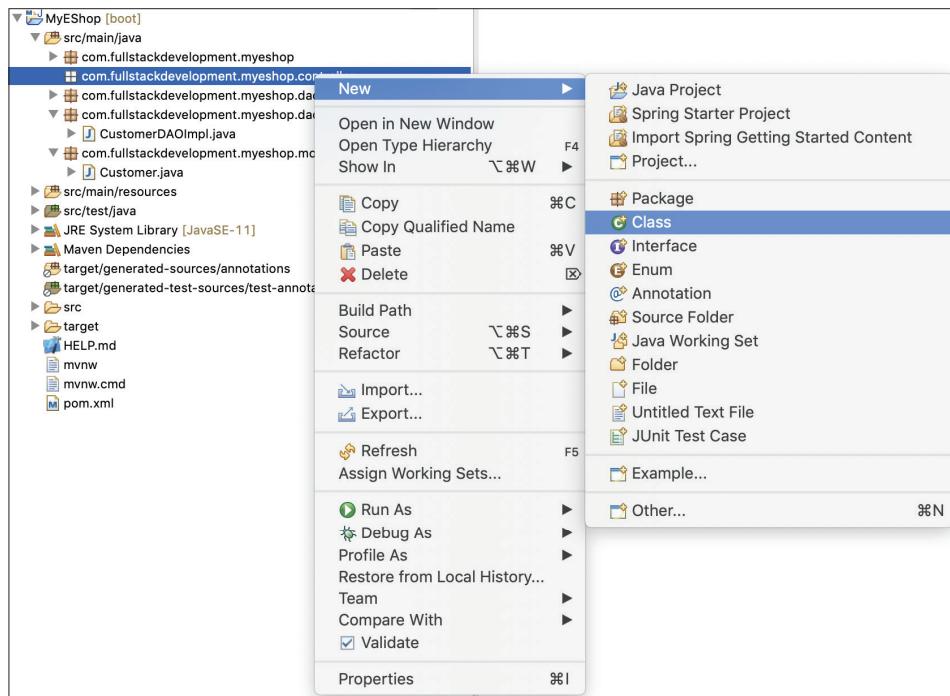
**Figure 22.34** New package detail window.

Once you click on the Finish button, it will create “com.fullstackdevelopment.myeshop.controller”, package which will be listed in the project explorer as shown in Figure 22.35.



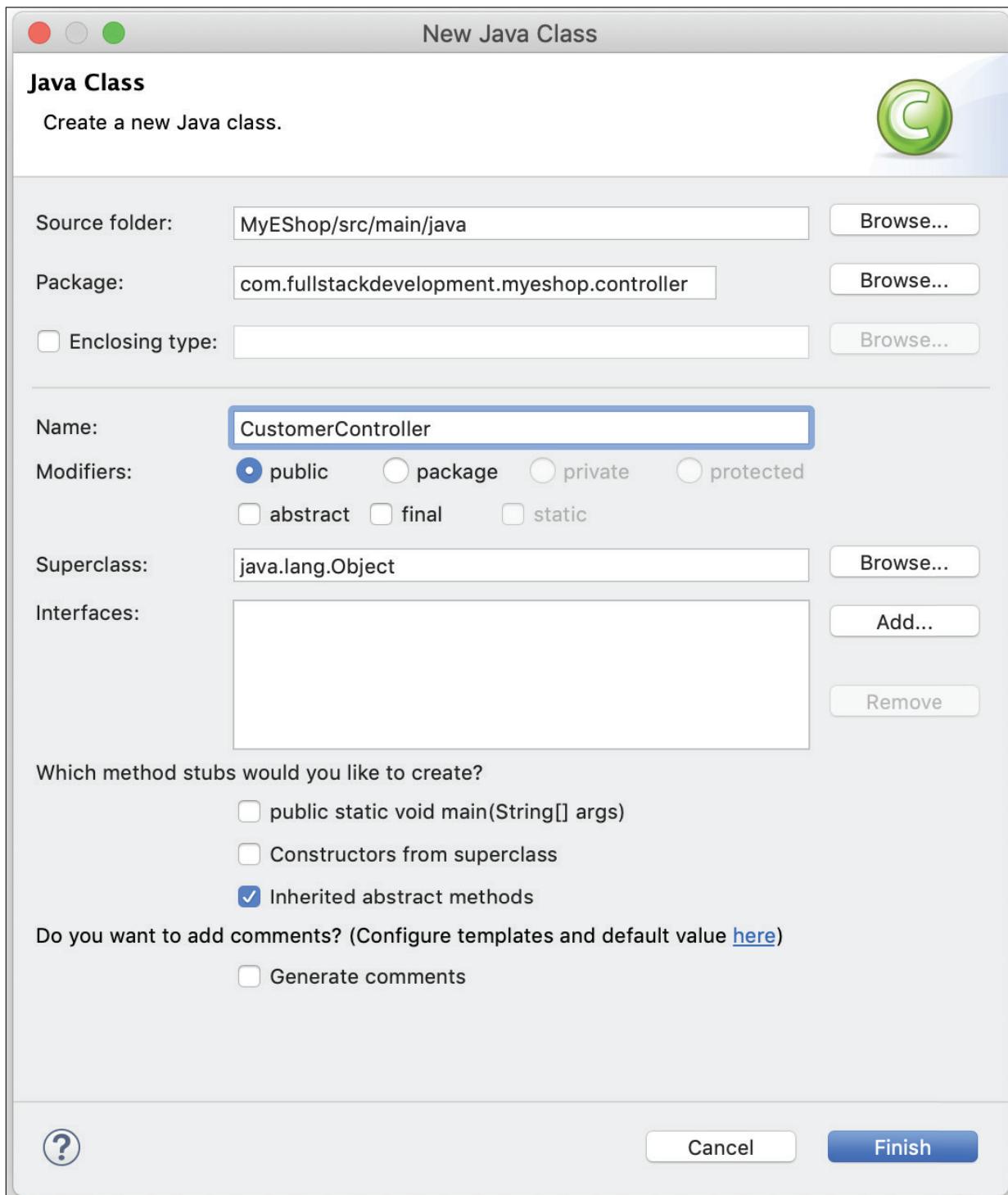
**Figure 22.35** Newly created package.

We can now add the CustomerController class in this package. You can add it in the same way we have added the earlier classes by right clicking on the package and selecting New->Class as shown in Figure 22.36.



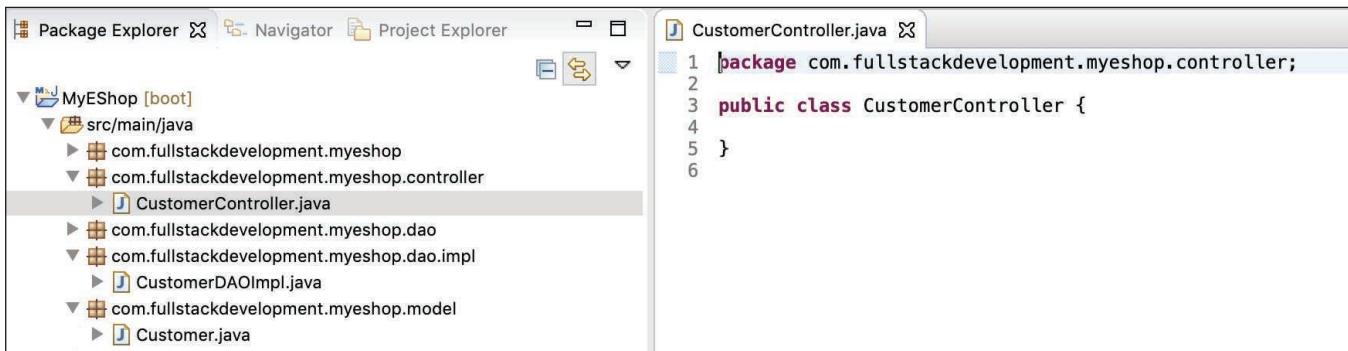
**Figure 22.36** Menu to create a new class.

Once you click on this menu, it will present a window to enter the class details. Enter “CustomerController” in the name field as shown in Figure 22.37.



**Figure 22.37** New Class Detail window.

Upon clicking on the Finish button, it will create a class named CustomerController under the package “com.fullstackdevelopment.myeshop.controller” as shown in Figure 22.38.



**Figure 22.38** Newly created Class Code.

Now, we can add the controller code in this class. In order to tell Spring that the class we are creating is a controller, we need to add `@Controller` annotation from "org.springframework.stereotype.Controller". See the following code in image below.

The screenshot shows the code editor with the title 'CustomerController.java'. The code has been updated to include the `@Controller` annotation:

```

1 package com.fullstackdevelopment.myeshop.controller;
2
3 import org.springframework.stereotype.Controller;
4
5 @Controller
6 public class CustomerController {
7
8 }
9

```

Since we will be accessing Customer database through the DAO we have created earlier, we need to access CustomerDAO object in the controller. So, let us declare the CustomerDAO object.

The screenshot shows the code editor with the title 'CustomerController.java'. The code now includes the declaration of a private CustomerDAO object:

```

1 package com.fullstackdevelopment.myeshop.controller;
2
3 import org.springframework.stereotype.Controller;
4
5 import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
6
7 @Controller
8 public class CustomerController {
9
10     private CustomerDAO customerDAO;
11
12 }
13

```

However, we have not initialized customerDAO object yet. We need to initialize it in order to use it in our code. Here, you can see how Spring uses Dependency Injection (DI) to get us the object we need without us initializing and managing its life cycle. We need to tell Spring to get us the CustomerDAO by using `@Autowired` annotation. This annotation tells Spring to manage the CustomerDAO object life cycle for us. See the following code in image below to know how to add `@Autowired` annotation.



```

1 package com.fullstackdevelopment.myeshop.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5
6 import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
7
8 @Controller
9 public class CustomerController {
10
11     @Autowired
12     private CustomerDAO customerDAO;
13
14 }
15

```

@Autowired annotation is located in “org.springframework.beans.factory.annotation.Autowired”. This gives us the object we need to access the Customer database via CustomerDAO. Now, Spring will take care of the life cycle and we do not have to worry about managing customerDAO object at all. We can simply access it and it will be available for us without us doing anything.

Now, let us add a controller method which will be an endpoint API for our REST web service. We will start with a simple step to understand what we are doing in order to create this endpoint. The first step is to create a method body. In our application, we will need to get customer data by using his/her email address. So, let us add this method.



```

1 package com.fullstackdevelopment.myeshop.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5
6 import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
7 import com.fullstackdevelopment.myeshop.model.Customer;
8
9 @Controller
10 public class CustomerController {
11
12     @Autowired
13     private CustomerDAO customerDAO;
14
15     public Customer findCustomerByEmail(String email) {
16         Customer customer = null;
17
18         return customer;
19     }
20 }
21

```

Please note that this is just a skeleton method we have added. We have not accessed the CustomerDAO yet to get the customer information. We have simply initialized an object and declared it null. Now, it is the time to access CustomerDAO to get the required data.

```

CustomerController.java ✎
1 package com.fullstackdevelopment.myeshop.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5
6 import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
7 import com.fullstackdevelopment.myeshop.model.Customer;
8
9 @Controller
10 public class CustomerController {
11
12     @Autowired
13     private CustomerDAO customerDAO;
14
15     public Customer findCustomerByEmail(String email) {
16
17         Customer customer = customerDAO.findCustomerByEmail(email);
18         return customer;
19
20     }
21 }
22

```

Here, we have replaced the null declaration with customerDAO.findCustomerByEmail(email). Since Spring is managing customerDAO object for us, we will simply get the data we need by executing this line of code. Now, you can see how easy it is to use DI in your code.

Also note that in this chapter we are not doing anything related to Hibernate, so the data will not come from the database yet. In Chapter 23, we will have the fully working method when we wire the database code using hibernate.

We are not done yet as Spring does not know if we want findCustomerByEmail(String email) to be an endpoint. We have to explicitly tell Spring by adding @RequestMapping annotation. This annotation maps HTTP request with our handler method, which turns into an endpoint. In @RequestMapping we can define the parameters we are expecting in the method. In our case, we are looking for email parameter. So, let us see how to add this to the annotation.

```

CustomerController.java ✎
1 package com.fullstackdevelopment.myeshop.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7
8 import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
9 import com.fullstackdevelopment.myeshop.model.Customer;
10
11 @Controller
12 public class CustomerController {
13
14     @Autowired
15     private CustomerDAO customerDAO;
16
17     @RequestMapping(value = "/customer/getCustomerByEmail/{email}", method = RequestMethod.GET)
18     public Customer findCustomerByEmail(String email) {
19
20         Customer customer = customerDAO.findCustomerByEmail(email);
21         return customer;
22
23     }
24 }
25

```

In the above image, you can see we have added `@RequestMapping(value = "/customer/getCustomerByEmail/{email}", method = RequestMethod.GET)` to define `findCustomerByEmail(String email)` as an endpoint which accepts email parameter as GET. In this example, “value = “/customer/getCustomerByEmail/{email}”,” defines the REST endpoint URL which will be used by our front end to access the Customer service. The next attribute `method = RequestMethod.GET` defines the request method we are using to get the parameter.

However, we are still not yet done as we need to map the incoming parameter to the parameter in the method. For this, we can use `@PathVariable("email")`. See the code in the following image.

```
J CustomerController.java & 
1 package com.fullstackdevelopment.myeshop.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.web.bind.annotation.PathVariable;
6 import org.springframework.web.bind.annotation.RequestMapping;
7 import org.springframework.web.bind.annotation.RequestMethod;
8
9 import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
10 import com.fullstackdevelopment.myeshop.model.Customer;
11
12 @Controller
13 public class CustomerController {
14
15     @Autowired
16     private CustomerDAO customerDAO;
17
18     @RequestMapping(value = "/customer/getCustomerByEmail/{email}", method = RequestMethod.GET)
19     public Customer findCustomerByEmail(@PathVariable("email") String email) {
20
21         Customer customer = customerDAO.findCustomerByEmail(email);
22         return customer;
23     }
24 }
25
26
```

## Summary

In this chapter, we have learned creating Spring MVC based REST web service. We have taken a step-by-step approach from setting up the development environment, creating Spring Boot project to creating endpoints for the front-end applications to consume. We have seen the use of Dependency Injection and how easy it is to let Spring manage the object lifecycle for us. We have explored the Data Access Object design pattern to create an interface for accessing database without exposing the underline persistence layer. We have also seen the use of creating a generic interface, which can be extended by all interfaces to get some generic methods to avoid duplication of code. In addition, we learned about a few useful annotations like `@Autowired`, `@RequestMapping`, and `@PathVariable`.

In Chapter 23, we will see some practical use of Hibernate to wire database operations to our model and DAO. You will explore new concepts such as `@OneToOne` and `@OneToMany` relationships, and CRUD operations such as `save`, `saveOrUpdate`, etc.

## Multiple-Choice Questions

1. Which of the following annotations is used to add dependency?  
 (a) `@RequestMapping`  
 (b) `@Entity`  
 (c) `@Autowired`  
 (d) `@PathVariable`
2. What is the full form of MVC?  
 (a) Model View Controller  
 (b) Model View Class  
 (c) Meta Vector Class  
 (d) Master View Class

3. Which of the following is a use of Package?
  - (a) To start a program
  - (b) Container for classes
  - (c) It looks good to start with package word
  - (d) To avoid a compiler error
4. Which of the following is a use of DAO?
  - (a) Provide access to an underlying database
  - (b) A class to contain business logic
  - (c) A class to contain API methods
  - (d) Just another class

## Review Questions

---

1. How do you download and install Spring Tool Suite IDE?
2. How do you create a new project in Spring Tool Suite IDE?
3. How do you create a model in Spring Tool Suite IDE?
4. Which annotation do you need to add on a class to tell Spring that it is a model?
5. How do you create Data Access Object Interface and implementation class?
6. Which annotation do you use to add on a class to tell Spring that it is a controller?
7. What is the role of a Data Access Object class?
8. What is the role of a model?
9. What is the role of a controller?
10. What is the use of `@Autowired` annotation?
11. What is the use of `@RequestMapping` annotation and how do you use it on a method?
12. How do you accept parameters in a controller method?
13. How do you specify if the method accepts GET or POST?
14. How do you accept the parameters in a controller method?

## Exercises

---

1. Create a simple service which can accept two integers and return their multiplication.
2. Define the use of `@Autowired`, `@RequestMapping`, and `@PathVariable`.

## Project Idea

---

Take an example of a restaurant table book application. Identify the required entities and APIs that you will need to allow users to book a table. Based on this, create model and

controller classes which can expose these APIs and connect to the database.

## Recommended Readings

---

1. Luciano Manelli. 2016. *Developing a Java Web Application in a Day: Step-by-step explanations with Eclipse, Tomcat and MySQL – A complete Java Project with Source Code (Java Web Programming Book 2)*. Luciano Manelli
2. Jeff Mcaffer. 2005. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java™ Applications*. Addison-Wesley: Massachusetts
3. John R Hines. 2016. *Eclipse 4 Introduction Course for Java Developer: A brief introduction to Java programming using Eclipse (Eclipse Development Book 2)*.



# Develop Models with Hibernate

## LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- How to wire hibernate to access database.
- How to perform CRUD operation on database with DAO.
- Basic yet important methods such as save and saveOrUpdate.

## 23.1 | Installing MySQL



We first need to create a database in which we can create the tables we need. For this example, we are using the MySQL database so you need to install it on your local computer. If you have not installed and setup MySQL, you can use the local server environment that will help you, and install and manage it with ease. Depending on your operating system, you could use one of the versions of MySQL explained in Sections 23.1.1 and 23.1.2. Please go through their documentation to learn about how to use them to set up MySQL.

### 23.1.1 Windows

**WampServer:** WampServer is the most useful local server environment for Windows users. It is free and opensource. It provides installation and management for Apache2, PHP, and MySQL.

Download: <http://www.wampserver.com/en/>  
Documentation: <http://www.wampserver.com/en/>

**HeidiSQL:** This is a simple-to-use MySQL client for windows which allows you to manage your database easily. It can connect to local as well as remote servers. HeidiSQL offers a lot of features and can connect to many database systems such as MariaDB, MySQL, Microsoft SQL, and PostgreSQL.

Download: <https://www.heidisql.com/download.php>  
Documentation: <https://www.heidisql.com/help.php>

### 23.1.2 Windows/Mac

**MAMP:** This is the Mac counterpart of the WampServer. Although different companies developed them, they have similar features in terms of managing local servers. MAMP is free to use. MAMP also has a Windows version so you could use this if you like this interface better than WampServer.

Download: <https://www.mamp.info/en/downloads/>  
 Documentation: <https://documentation.mamp.info/>

Once you setup MySQL on your local machine, if you are not comfortable using command line, you may use one of the following free graphical user interface (GUI) applications to create database and tables.

**Sequel pro:** This is a free to use MySQL client. It offers a simple to use interface and good features. Although it does not offer the most advanced features, it is alright for small projects.

Download: <https://sequelpro.com/download>  
 Documentation: <https://sequelpro.com/docs>

## 23.2 | Create Database and Tables



After setting up MySQL and your desired GUI, it is time to create our database. For our eShop example, we will give the same name to our database as our application. So create a database using the GUI and give it a name “MyEShop” as shown in Figure 23.1.

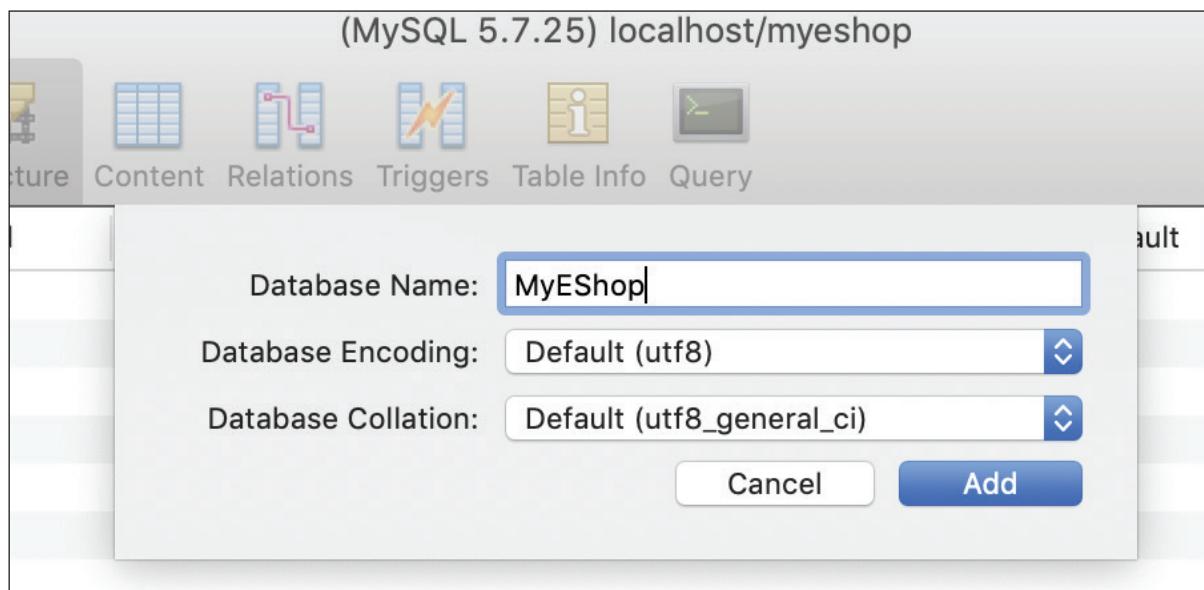
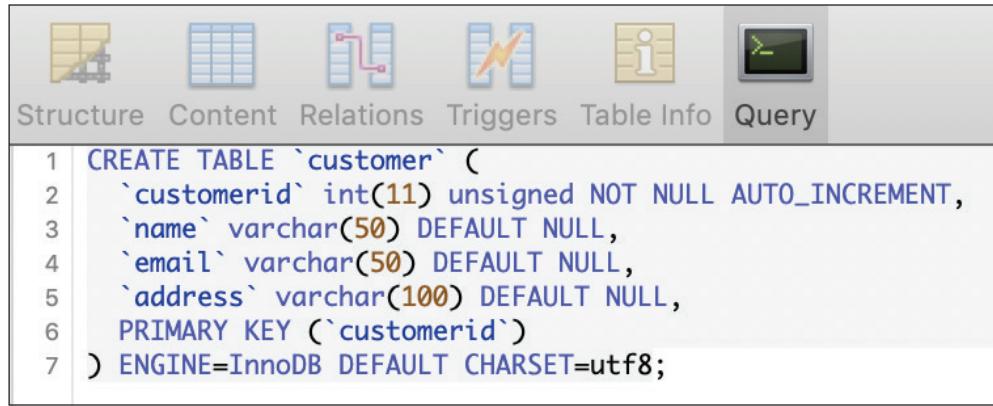


Figure 23.1 Adding a new database in GUI.

Once you click on the Add button, GUI will create the database for you. Now, it is time to add a table. You can either use the GUI to add it or use the following SQL which you can run in the Query window to create the Customer table with the fields we need.

```
CREATE TABLE `customer` (
  `customerid` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(50) DEFAULT NULL,
  `email` varchar(50) DEFAULT NULL,
  `address` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`customerid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

See the following screenshot in Figure 23.2 to see how to run this query in the GUI.



```

CREATE TABLE `customer` (
  `customerid` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(50) DEFAULT NULL,
  `email` varchar(50) DEFAULT NULL,
  `address` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`customerid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

**Figure 23.2** Query window to run CREATE TABLE command.

Please note that every table should have id column so we can identify the record and link it to other tables as foreign key. In this case, we will be more specific and call our id column as “customerid”. This is the field we have not added in our Customer model yet so we need to update our model to accommodate this field.

### 23.2.1 Linking Tables to Models

We will add customerid field as shown in the code in the following image, and let Spring know that this is an id by adding @Id annotation from javax.persistence package.



```

package com.fullstackdevelopment.myeshop.model;

import java.sql.Timestamp;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="customer")
public class Customer {

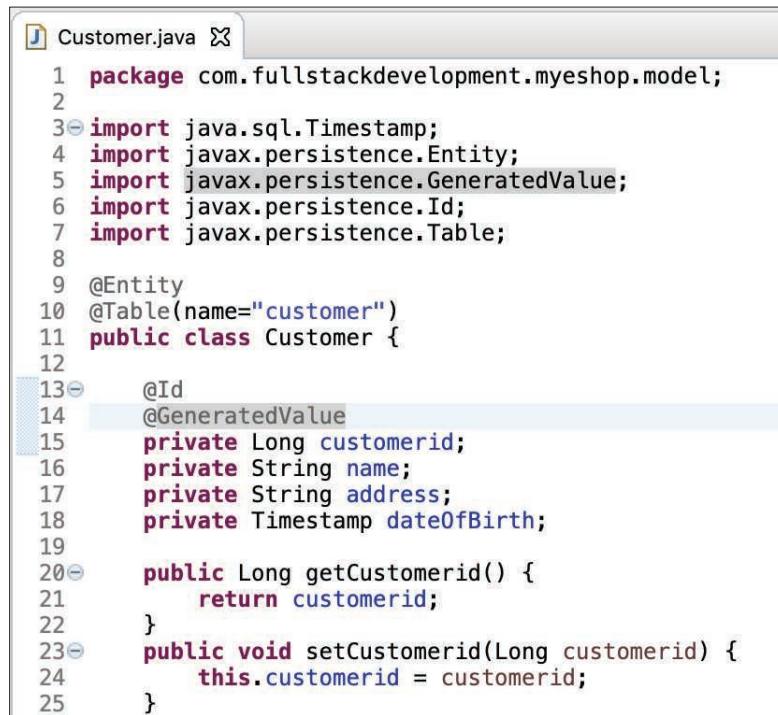
    @Id
    private Long customerid;
    private String name;
    private String address;
    private Timestamp dateOfBirth;

    public Long getCustomerid() {
        return customerid;
    }

    public void setCustomerid(Long customerid) {
        this.customerid = customerid;
    }
}

```

Since this is an id column, we would like it to be incremental. This is what we need to tell Hibernate as well. For this, we will be using @GeneratedValue annotation from javax.persistence package.



```

1 package com.fullstackdevelopment.myeshop.model;
2
3 import java.sql.Timestamp;
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.Id;
7 import javax.persistence.Table;
8
9 @Entity
10 @Table(name="customer")
11 public class Customer {
12
13     @Id
14     @GeneratedValue
15     private Long customerid;
16     private String name;
17     private String address;
18     private Timestamp dateOfBirth;
19
20     public Long getCustomerid() {
21         return customerid;
22     }
23     public void setCustomerid(Long customerid) {
24         this.customerid = customerid;
25     }

```



How do the values in customerid table get stored if @GeneratedValue annotation is removed?

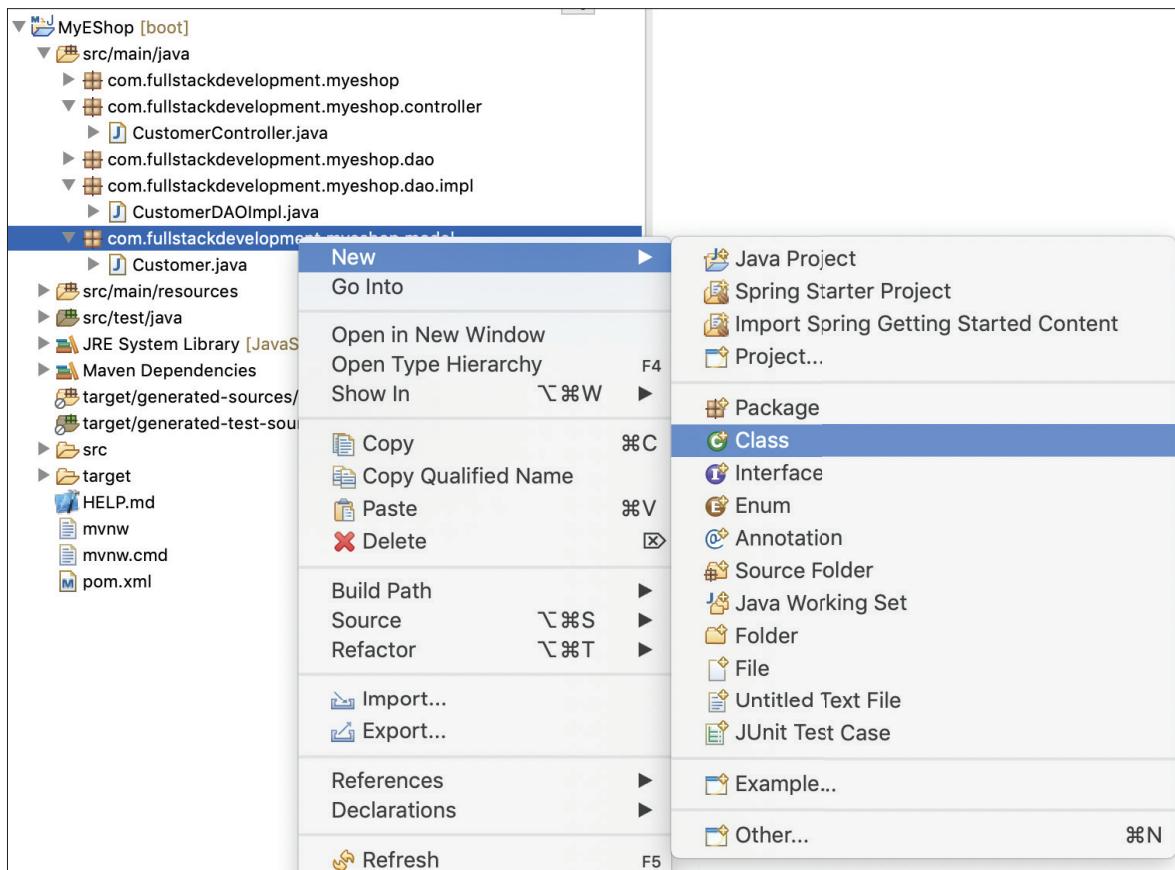
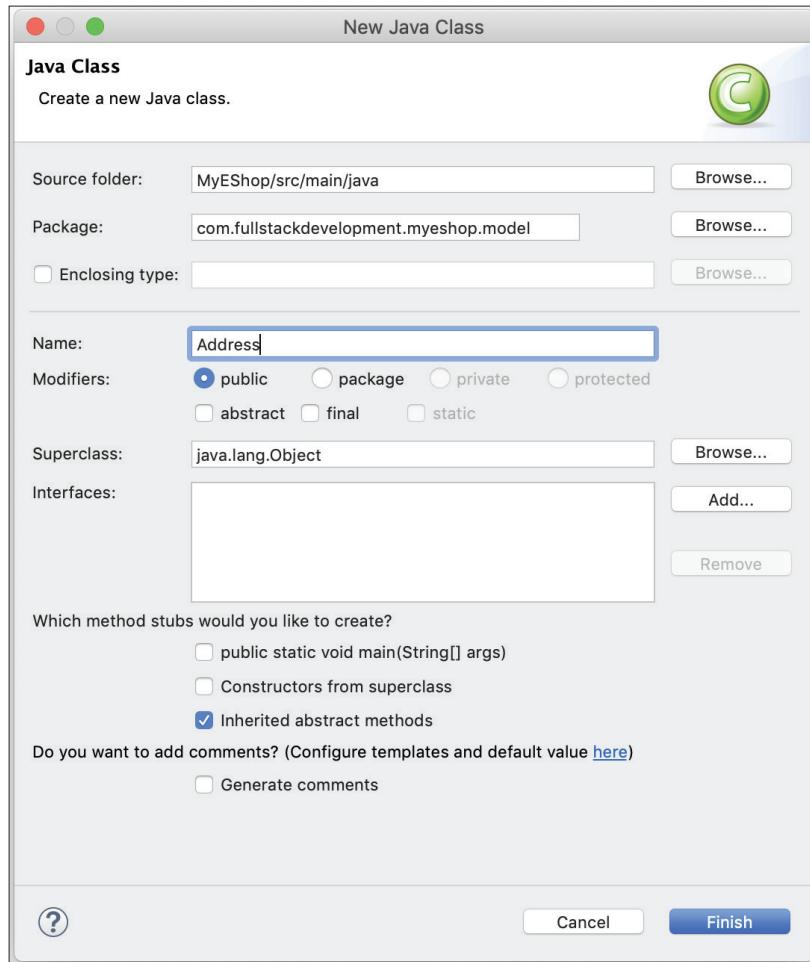


Figure 23.3 Menu to create a new class.

There is one more important step to map this model to the customer table we have created. This step we have already done in Chapter 22 by adding `@Table` annotation to our `Customer` class name. We also specified the table name that is mapped to this model by using name attribute `@Table(name="customer")`.

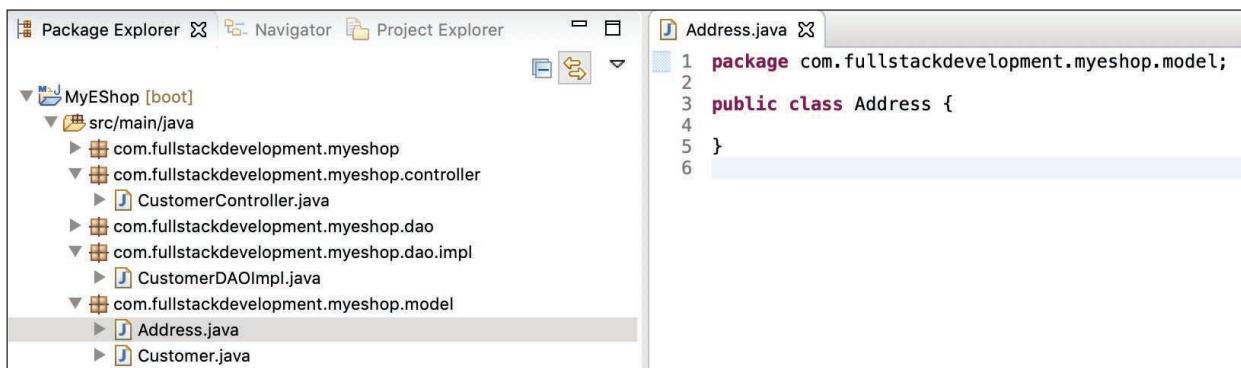
In Chapter 22 we talked about making address as a separate entity so we can store multiple addresses for the customer. So, let us work on that now. We need to update our `Customer` table and model for this. Let us first create a new model called `Address`, with basic address related fields. Right-click on the “`com.fullstackdevelopment.myeshop.model`” package and click on `New->Class`.

After clicking on this menu, you will see a window to add details about the model. Let us call this class as “`Address`” as shown in Figure 23.4.



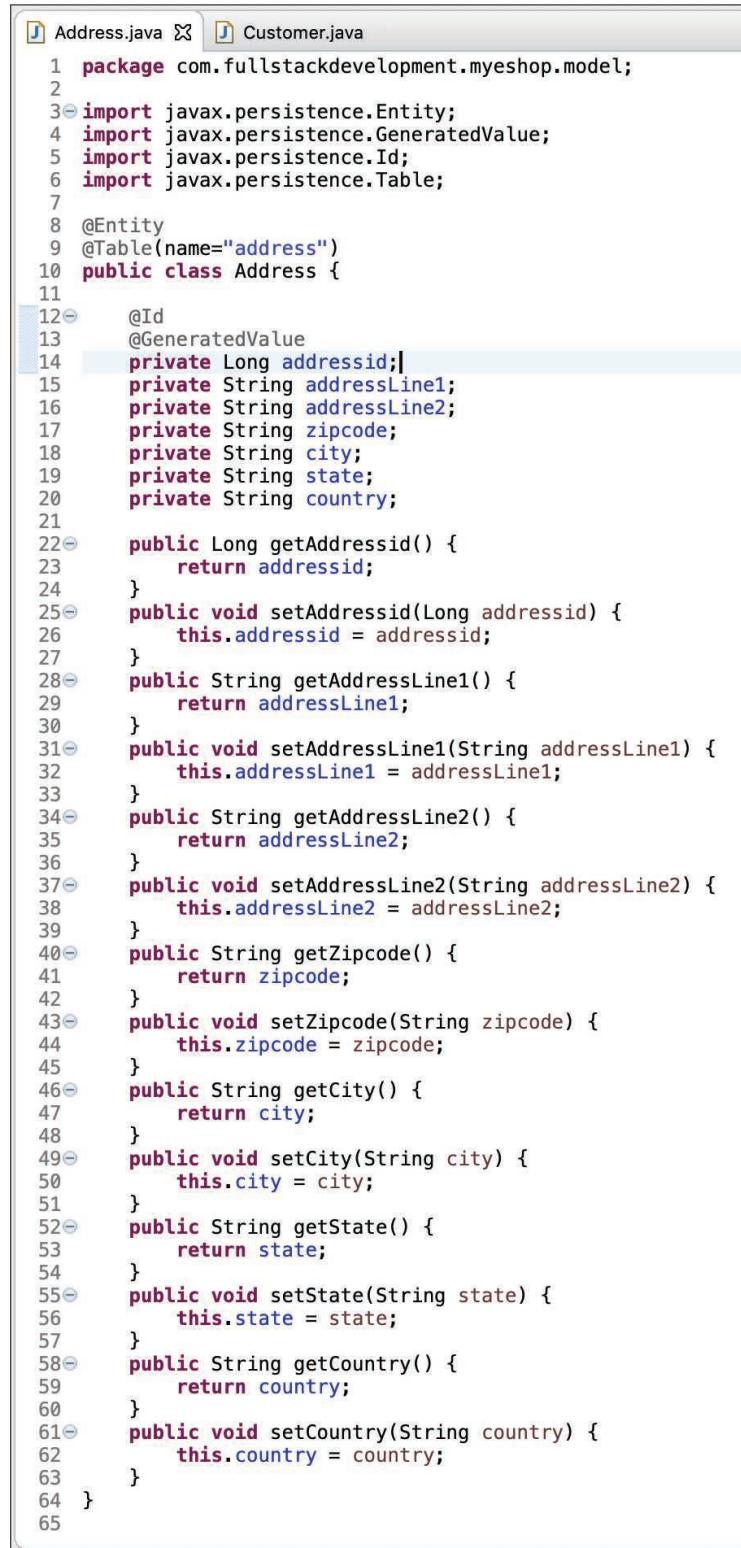
**Figure 23.4** New class detail window.

Upon clicking on the `Finish` button, it will create the `Address` class under “`com.fullstackdevelopment.myeshop.model`” package as shown in Figure 23.5.



**Figure 23.5** Newly added `Address` class.

Once the class is created, add basic address related fields and “addressid” as an id field with the required annotations.



```

1 package com.fullstackdevelopment.myeshop.model;
2
3 import javax.persistence.Entity;
4 import javax.persistence.GeneratedValue;
5 import javax.persistence.Id;
6 import javax.persistence.Table;
7
8 @Entity
9 @Table(name="address")
10 public class Address {
11
12     @Id
13     @GeneratedValue
14     private Long addressid;
15     private String addressLine1;
16     private String addressLine2;
17     private String zipcode;
18     private String city;
19     private String state;
20     private String country;
21
22     public Long getAddressid() {
23         return addressid;
24     }
25     public void setAddressid(Long addressid) {
26         this.addressid = addressid;
27     }
28     public String getAddressLine1() {
29         return addressLine1;
30     }
31     public void setAddressLine1(String addressLine1) {
32         this.addressLine1 = addressLine1;
33     }
34     public String getAddressLine2() {
35         return addressLine2;
36     }
37     public void setAddressLine2(String addressLine2) {
38         this.addressLine2 = addressLine2;
39     }
40     public String getZipcode() {
41         return zipcode;
42     }
43     public void setZipcode(String zipcode) {
44         this.zipcode = zipcode;
45     }
46     public String getCity() {
47         return city;
48     }
49     public void setCity(String city) {
50         this.city = city;
51     }
52     public String getState() {
53         return state;
54     }
55     public void setState(String state) {
56         this.state = state;
57     }
58     public String getCountry() {
59         return country;
60     }
61     public void setCountry(String country) {
62         this.country = country;
63     }
64 }
65

```

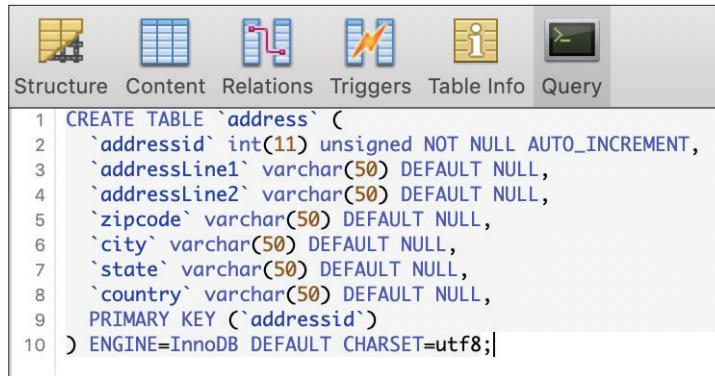
As you can see, we have added the basic fields and `@Entity` annotation along with `@Table` annotation and specified our table as “address”. Now, let us create this table in the database as well. You can run the following SQL in the Query window of the GUI.

```

CREATE TABLE `address` (
    `addressid` int(11) unsigned NOT NULL AUTO_INCREMENT,
    `addressLine1` varchar(50) DEFAULT NULL,
    `addressLine2` varchar(50) DEFAULT NULL,
    `zipcode` varchar(50) DEFAULT NULL,
    `city` varchar(50) DEFAULT NULL,
    `state` varchar(50) DEFAULT NULL,
    `country` varchar(50) DEFAULT NULL,
    PRIMARY KEY (`addressid`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

```

See the following image to see how to run this query.



#### QUICK CHALLENGE

Based on the entity relationship diagram, create SQL for other tables.

### 23.2.2 Setting up Relationships

Now, we need to link this table with Customer so Customer can store multiple addresses. For this, we need to update our SQL as well as our Model code. Let us first update our table to link address to customer. We can use “customerid” as a foreign key in the address table.

First add “customerid” as a field in the “address” table. You can use the following command for that.

```
ALTER TABLE `address` ADD COLUMN customerid int(11) not null;
```

Please note that we are not yet adding a database constraint as foreign key on the address table. We are just creating customerid field in the address table so hibernate can link these two tables for us.

Now, we need to update our Address class to add this newly added field. However, we will not add this field as Long but we will reference it to the Customer object. Later, we will see how to link these tables in the model code.

```

8  @Entity
9  @Table(name="address")
10 public class Address {
11
12     @Id
13     @GeneratedValue
14     private Long addressid;
15     private String addressLine1;
16     private String addressLine2;
17     private String zipcode;
18     private String city;
19     private String state;
20     private String country;
21     private Customer customer;
22

```

You will also need to add Getter and Setter for this field.

```

65④  public Customer getCustomer() {
66      return customer;
67  }
68④  public void setCustomer(Customer customer) {
69      this.customer = customer;
70  }

```

Now, it is time to let Spring know how we need to access Customer and Address. As discussed earlier, every customer can have multiple addresses such as Shipping, Billing, etc. Hence, the customer can have one-to-many relationship with address. Similarly, address can have many-to-one relationship with customer. This is what we need to define in the code so Spring will know how to treat these two entities.

Before we specify in both the models, we first need to add address field in the Customer model. Since we would like to define one-to-many relationship, we will add address as List so we can get multiple addresses for one customer entry. See the code in the following image.

```

11 @Entity
12 @Table(name="customer")
13 public class Customer {
14
15④  @Id
16  @GeneratedValue
17  private Long customerid;
18  private String name;
19  private String address;
20  private Timestamp dateOfBirth;
21  private List<Address> addresses;
22
23④  public List<Address> getAddresses() {
24      return addresses;
25  }
26④  public void setAddresses(List<Address> addresses) {
27      this.addresses = addresses;
28  }

```

Once we define this, we now need to define these relationships. For this, we can use @OneToMany and @ManyToOne annotations. See the Customer model code in the following image.

```

1 package com.fullstackdevelopment.myeshop.model;
2
3④ import java.sql.Timestamp;
4 import java.util.List;
5
6 import javax.persistence.CascadeType;
7 import javax.persistence.Entity;
8 import javax.persistence.FetchType;
9 import javax.persistence.GeneratedValue;
10 import javax.persistence.Id;
11 import javax.persistence.OneToMany;
12 import javax.persistence.Table;
13
14 @Entity
15 @Table(name="customer")
16 public class Customer {
17
18④  @Id
19  @GeneratedValue
20  private Long customerid;
21  private String name;
22  private String address;
23  private Timestamp dateOfBirth;
24
25④  @OneToMany(mappedBy="customer",fetch = FetchType.LAZY, cascade = CascadeType.MERGE)
26  private List<Address> addresses;
27

```

@OneToMany annotation has mappedBy property where we can define the variable reference from the Address class. Earlier, we have added field **private Customer customer;** in the address class. This is what we need to reference here.



What will happen if we remove @OneToMany annotation from the addresses field?

Then we have another property called fetch. This is an important property to set as it defines FetchType. In our case, we have used LAZY as the fetch type. It means Hibernate will not query tables until we ask for it specifically. This will speed up the fetch operation as Hibernate does not need to define join while running select query on the Customer table. If we explicitly ask to give customer addresses then Hibernate will add the join query. This helps in increasing application performance. The other fetch type is called EAGER. This fetch type instructs Hibernate to get addresses each time we ask for a customer object. It adds a little penalty on the performance side.

The last property that you see is “cascade”. This property defines what happens when a CRUD operation is performed on customer object. There are many cascade types you can use, as follows.

1. **CascadeType.PERSIST:** When used, `save()` or `persist()` operations cascade to related entities.
2. **CascadeType.MERGE:** When used on the owning entity, all the related entities of the owning entity are merged upon the owning entity is merged.
3. **CascadeType.REFRESH:** When used, related entities are refreshed.
4. **CascadeType.REMOVE:** When used, upon deletion of the owning entity, all the related entities get removed.
5. **CascadeType.DETACH:** When used, in case of “manual detach” all the related entities are detached.
6. **CascadeType.ALL:** When used, it performs all the above cascade operations.

In order to get it working properly, we also need to tell Hibernate about Address to Customer relationship. Hence, we need to define @ManyToOne relationship on the customer object in the Address class.

```

Address.java ✘ Customer.java
1 package com.fullstackdevelopment.myeshop.model;
2
3 import javax.persistence.CascadeType;
4 import javax.persistence.Entity;
5 import javax.persistence.FetchType;
6 import javax.persistence.GeneratedValue;
7 import javax.persistence.Id;
8 import javax.persistence.JoinColumn;
9 import javax.persistence.ManyToOne;
10 import javax.persistence.Table;
11 import com.fasterxml.jackson.annotation.JsonBackReference;
12
13 @Entity
14 @Table(name="address")
15 public class Address {
16
17     @Id
18     @GeneratedValue
19     private Long addressid;
20     private String addressLine1;
21     private String addressLine2;
22     private String zipcode;
23     private String city;
24     private String state;
25     private String country;
26
27     @JsonBackReference
28     @ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.MERGE)
29     @JoinColumn(name="customerid", referencedColumnName = "customerid")
30     private Customer customer;
31

```

As you can see, we have added the following three lines on the Customer object declaration.

```
@JsonBackReference
@ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.MERGE)
@JoinColumn(name="customerid", referencedColumnName = "customerid")
```

This will tell Hibernate how to link address with customer.

We have already learned fetch and cascade, so let us jump on to the other two annotations. `@JoinColumn` annotation tells Hibernate about the column in the database that we need use for the link. This is the foreign key field we have created in the address table. The name property allows us to specify this field name from the table, in our case it is “customerid”. The `referencedColumnName` property allows us to specify the column name (primary key) of the customer class, which is linked with address via “customerid” field. There is one more annotation `@JsonBackReference` you can see on top of `@ManyToOne` annotation. This annotation is designed to solve the infinite recursion problem by linking two-way – one for Parent and other for Child.

Now, let us look at another type of relationship. When two entities are related to each other in a singular fashion, it is known as one-to-one relationship. This type of relationship is defined by `@OneToOne` annotation. The owning class field will have this annotation.

#### QUICK CHALLENGE

State the differences of `@OneToOne`, `@OneToMany`, and `@ManyToOne`.

### 23.3 | Making DAO to Perform CRUD



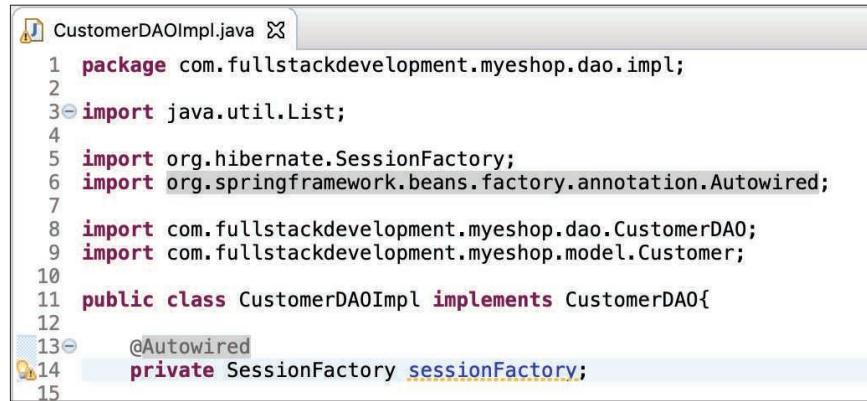
We can now move on to the DAO class section to write code that can perform CRUD operations on these models.

In our `CustomerDAOImpl` class, we will be adding this interface code, which will hide the model details and only expose it via `CustomerDAO` interface methods and hence further hides the implantation from the calling classes. Since we will be performing CRUD operations, we need to make sure the transactions are handled properly. For this, we can simply annotate the class with `@Transactional` annotation. This tells Spring that we need Spring's help to manage the transactions for this class.

```
CustomerDAOImpl.java
1 package com.fullstackdevelopment.myeshop.dao.impl;
2
3 import java.util.Collection;
4 import java.util.Optional;
5
6 import javax.transaction.Transactional;
7
8 import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
9 import com.fullstackdevelopment.myeshop.model.Customer;
10
11 @Transactional
12 public class CustomerDAOImpl implements CustomerDAO{
13 }
```

If we asked Spring to manage transactions on our behalf by annotating the class as `@Transactional`, Spring will create a proxy class which is invisible at runtime and provides a way to inject behavior into the object before, after, or around method calls.

Now, we can move on to write code for `findCustomerByEmail(String email)` method. In order to use Hibernate to perform CRUD operations, we need `SessionFactory` from the Hibernate package. As shown in the code in the following image, we will use `@Autowired` annotation to inject `SessionFactory` to `CustomerDAOImpl` class.



```

1 package com.fullstackdevelopment.myeshop.dao.impl;
2
3 import java.util.List;
4
5 import org.hibernate.SessionFactory;
6 import org.springframework.beans.factory.annotation.Autowired;
7
8 import com.fullstackdevelopment.myeshop.dao.CustomerDAO;
9 import com.fullstackdevelopment.myeshop.model.Customer;
10
11 public class CustomerDAOImpl implements CustomerDAO{
12
13     @Autowired
14     private SessionFactory sessionFactory;
15

```

We also need to create a constructor and pass SessionFactory parameter, which will be configured in the XML configuration.

```

public CustomerDAOImpl(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
}

```

Once we get the sessionFactory object, we can add the following code to fetch a Customer record by email. Before we proceed with this task, we need to add the email field in the Customer model class. We have already added this into the database, so it is time to add in the model.



```

1 package com.fullstackdevelopment.myeshop.model;
2
3 import java.sql.Timestamp;
4
5
6 @Entity
7 @Table(name="customer")
8 public class Customer {
9
10     @Id
11     @GeneratedValue
12     private Long customerid;
13     private String name;
14     private String email;
15     private Timestamp dateOfBirth;
16
17
18
19
20
21
22
23

```

As you can see from the code in the above image, we have removed the String type “address” field and added String type “email” field. We have also created getter and setter for this field. Let us proceed with adding customer search code with email address.



```

51
52     @Override
53     public Customer findCustomerByEmail(String email) {
54         CriteriaBuilder criteriaBuilder = sessionFactory.getCurrentSession().getCriteriaBuilder();
55         CriteriaQuery<Customer> criteriaQuery = criteriaBuilder.createQuery(Customer.class);
56         Root<Customer> root = criteriaQuery.from(Customer.class);
57         criteriaQuery.select(root).where(criteriaBuilder.equal(root.get("email"), email));
58         Query<Customer> q = sessionFactory.getCurrentSession().createQuery(criteriaQuery);
59         return q.getSingleResult();
60     }

```

In the above example, we have used CriteriaBuilder from “javax.persistence.criteria.CriteriaBuilder” package. It is accessible via sessionFactory. This CriteriaBuilder object is then used to get CriteriaQuery instance. Then the “from” method is used to set the Customer class as “root” on this query object. Now the root is set, we used “where” method to set the comparison on email column. Then we used create query object for this CriteriaQuery that we built. And finally, we run “q.getSingleResult()” to get the desired record for the email parameter we are passing to this method.

**QUICK  
CHALLENGE**

With help of findCustomerByEmail(String email) example, write code for get(int id) and getAll methods.

Let us move on to write code for the save method. In Hibernate, there are three options we can use to save the record in the database.

1. **save():** As the name suggests, this is used to save an entity into the database. It also returns a generated identifier. If the entity already exists in the database, it throws an exception.
2. **persist():** It is very similar to save method. The only difference is it does not return a generated identifier but returns a void.
3. **saveOrUpdate():** This is useful when you do not know if the entity exists in the database. This method will either save or update the entity into the database. Hence, it does not throw an exception as `save()` method throws if the entity already exists in the database; it will simply update it.

The code in the following image shows how to save record through Hibernate and how easy it is to set data without writing a single line of SQL.

```

@Override
public Long save(Customer t) {
    Customer customer = t;

    if (customer == null) {
        customer = new Customer();
        customer.setName("Albert Einstein");
        customer.setEmail("Albert@Einstein.com");

        // set date of birth
        SimpleDateFormat format = new SimpleDateFormat("yyyyMMdd");
        try {
            Date parsed = format.parse("1879014");
            Timestamp dateOfBirth = new Timestamp(parsed.getTime());
            customer.setDateOfBirth(dateOfBirth);
        } catch (ParseException e) {
            System.out.println("Error in parsing the date of birth");
            e.printStackTrace();
        }
    }

    // set billing and shipping addresses
    List<Address> allAddresses = new ArrayList<Address>();

    Address shippingAddress = new Address();
    shippingAddress.setAddressLine1("112");
    shippingAddress.setAddressLine2("Mercer St");
    shippingAddress.setCity("Princeton");
    shippingAddress.setZipcode("NJ 08540");
    shippingAddress.setCountry("USA");
    sessionFactory.getCurrentSession().save(shippingAddress);
    allAddresses.add(shippingAddress);

    Address billingAddress = new Address();
    billingAddress.setAddressLine1("112");
    billingAddress.setAddressLine2("Mercer St");
    billingAddress.setCity("Princeton");
    billingAddress.setZipcode("NJ 08540");
    billingAddress.setCountry("USA");
    sessionFactory.getCurrentSession().save(billingAddress);
    allAddresses.add(billingAddress);

    customer.setAddresses(allAddresses);
}

sessionFactory.getCurrentSession().save(customer);
sessionFactory.getCurrentSession().flush();

return customer.getCustomerId();
}

```

In the above code, multiple steps are taking place. We are first checking if the parameter is null or not. If it is, say, null, we are initializing a Customer object. Now, we will use this customer object to set field values. If it is not null then we are directly saving it using the sessionFactory. In the null case, setting up name and email are straightforward processes. For setting up date of birth, we need to convert the given date into a Timestamp with the help of SimpleDateFormat. After that, we are declaring an ArrayList for Address entity. This ArrayList will hold all the addresses. We are creating a shippingAddress by initializing an Address object and setting up the field data. After that we are saving the address entity into the database with the help of sessionFactory. We are repeating the process for the billingAddress. Once the addresses are saved, we are adding this list to the customer object's addresses field and later saving it into the database. In the end, we are flushing the database so database is updated. Since `save()` method returns generatedId (in our case customerid), we are simply returning it.

**QUICK CHALLENGE**

By using save method example shown in Section 23.3, write code for update method.

Now, let us write code for “delete” method. There are two types of entities we will be dealing with when deleting from the database. One is transient and another one is persistence. Since the transient entity is not associated with the session, we use identifier to remove the instance from the database. For example, if Customer object is transient then we can delete it by specifying ID of it. See the following code block.

```
Customer customer = new Customer();
customer.setCustomerid(1L);
sessionFactory.getCurrentSession().delete(customer);
```

The above code will delete a Customer record which contains customerid = 1.

In case of persistence, we load the entity first and then delete it. We use sessionFactory object to load the entity.

```
Customer customer = sessionFactory.getCurrentSession().load(Customer.class, 1L);
if(customer != null) {
    sessionFactory.getCurrentSession().delete(customer);
}
```

In the above example, we are first loading Customer using sessionFactory and then making sure it is not null before executing delete operation on it.

In our case, since we are getting a customer object via parameter, we do not need to load this entity. We will just make sure that the entity is not null before running the delete operation on it.

```
@Override
public void delete(Customer t) {
    Customer customer = t;
    if(customer != null) {
        sessionFactory.getCurrentSession().delete(customer);
    }
}
```

## Summary

In this chapter, we have learned how to install and use MySQL, create database and tables, and run queries using GUI. We have also seen how to use Hibernate to perform CRUD operations on a database. We have learned various relationships like @OneToOne and @OneToMany. We have also learned about performing CRUD operations on a database including the difference between save, persist, and saveOrUpdate.

## Multiple-Choice Questions

---

1. What is RDBMS?
    - (a) Relational Database Management System
    - (b) Rational Data Batch Management System
    - (c) Relational Database Messaging System
    - (d) Rotational Database Management System
  2. Which of the following is not a valid cascade type?
    - (a) CascadeType.PERSIST
    - (b) CascadeType.MERGE
    - (c) CascadeType.SAVE
    - (d) CascadeType.REFRESH
  3. `save()` is used to update a record in Hibernate.
- (a) True
  - (b) False
  4. SessionFactory is a thread safe object
    - (a) True
    - (b) False
  5. \_\_\_\_\_ annotation is used to make value of id column incremental.
    - (a) `@GeneratedValue`
    - (b) `@Id(Increment)`
    - (c) `@Id(ColumnValue = "Increment")`
    - (d) `@Incremental`

## Review Questions

---

1. What is MySQL?
2. What is the use of Database?
3. What is the difference between DBMS and RDBMS?
4. How to make sure that the class is treated as model to map with a table in the database?
5. What is Hibernate?
6. What is object-relational mapping?
7. What are the disadvantages of using object-relational mapping?
8. What is the difference between `save()` and `saveOrUpdate()`?
9. When can you use `persist()`?
10. What is CriteriaBuilder?

## Exercises

---

1. Define all the different Database encodings and their effect on table.
2. Define all the different Database collations and their effect on table.
3. Change the column name “name” to “fullname” of the table customer we created in section 23.2.
4. Explain the use of `@JsonBackReference` and the effect on the table if removed from the field.

## Project Idea

---

Create a simple application to create a TODO list application. Use Hibernate to store and retrieve data. Define all the relationships and make sure you use `@OneToMany` and `@OneToOne` relations.

## Recommended Readings

---

1. Thorben Janssen and Steve Ebersole. 2017. *Hibernate Tips: More Than 70 Solutions to Common Hibernate Problems*. Createspace Independent Pub
2. K. Santosh Kumar. 2017. *Spring and Hibernate*. McGraw Hill: New York
3. Hibernate User Guide – [https://docs.jboss.org/hibernate/orm/5.2/userguide/html\\_single/Hibernate\\_User\\_Guide.html](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html)
4. Hibernate ORM Guide – <https://hibernate.org/orm/documentation/5.4/>