CHAPTER 4

Java Syntax

Languages are means of communication—verbal or written—between people. Whether they are natural or artificial, they are made of terms and have rules on how to use them to perform the task of communication. Programming languages are means of communicating with a computer. The communication with a computer is a written communication; basically, the developer defines some instructions to be executed, communicates them through an intermediary to the computer, and if the computer understands them, performs the set of actions, and depending on the application type, some sort of reply is returned to the developer.

In the Java language, communication is done through an intermediary—the Java virtual machine. The set of programming rules that define how terms should be connected to produce an understandable unit of communication is called **syntax**. Java borrowed most of its syntax from a programming language called C++, which has a syntax based on the C language. C syntax borrows elements and rules from languages that preceded it, but in essence, all of these languages are based on the natural English language.

Maybe Java got a little cryptic in version 8 because of the introduction of lambda expressions, but when writing a Java program, if you are naming your terms properly in the English language, the result should be code that is easily readable, like a story.

A few details were covered in **Chapter** 3; packages and modules were covered enough to give you a solid understanding of their purpose to avoid confusion with the organization of the project and aimless fumbling through the code. But as expected when it comes to actual code writing, the surface has been barely scratched. Thus, let's begin our deep dive into Java.

I. Cosmina, *Java for Absolute Beginners*, https://doi.org/10.1007/978-1-4842-3778-6_4

Base Rules of Writing Java Code

Before writing Java code, let's go over a few rules that you should follow to make sure your code actually works. Let's depict the class we ended **Chapter** 3 with by adding a few details.

```java
package com.apress.bgn.ch3.helloworld;

import java.util.List;

/**
* this is a JavaDoc comment
*/
public class HelloWorld {
   public static void main(String... args) {
        //this is a one-line comment
        List<String> items = List.of("1", "a", "2", "a", "3", "a");
       items.forEach(item -> {
        /* this is a
                multi-line
           comment */
            if (item.equals("a")) {
               System.out.println("A");
            } else {
               System.out.println("Not A");
            }
       });
   }
}
```

Next, I'll cover each rule in its own section.

Package Declaration

A Java file always starts with the **package declaration**. The package name can contain letters and numbers, separated by dots. Each part matches a directory in the path to the classes contained in it. The package declaration should reveal the name of the application and the purpose of the classes in the package. Let's take the package naming used for the sources of this book: `com.apress.bgn.ch4.basic`. If we split the package name in pieces, the meaning of each piece is described as follows.

- `com.apress` is the domain of the application, or who owns the application in this case
- `bgn` is the scope of the code, in this case the book it is written for (Java for Absolute **Beginn**ers)
- `ch4` is the purpose of the classes in Chapter 4
- `basic` is a more refined level of the purpose for the classes, these classes are simple, used to depict basic Java notions

Import Section

The **import section** follows the package declaration. This section contains the fully qualified names of all classes, interfaces, and enums used within the file. Look at the following code sample.

```
package java.lang;

import java.io.Serializable;
import java.io.ObjectStreamField;
import java.io.UnsupportedEncodingException;
import java.lang.annotation.Native;
import java.nio.charset.Charset;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Formatter;
import java.util.Locale;
...
```

```
public final class String
    implements Serializable, Comparable<String>, CharSequence {

    private  static  final  ObjectStreamField  serialPersistentFields  =
        new ObjectStreamField0;
    ...
}
```

It is a snippet from the official Java String class. Every import statement makes reference to the package and the name of a class used within the String class body.

Special import statements import static variables and static methods. Static variables and methods can be used without the need to instantiate a class. In the JDK, there is a class used for mathematical processes. It contains static variables and methods that can be used by developers to implement code that solves mathematical problems. Look at the following code.

```
package com.apress.bgn.ch4.basic;

import static java.lang.Math.PI;

import static java.lang.Math.sqrt;

public class Sample extends Object {
    public static void main(String... args) {
        System.out.println("PI value =" + PI);

        double result = sqrt(5.0);

        System.out.println("SQRT value =" + result);
    }
}
```

By putting import and static together, we can declare a fully qualified name of a class and the method or the variable we are interested in using in the code. This allows us to use the variable or method directly, without the name of the class it is declared in. Without the static imports, the code has to be rewritten like this:

```
package com.apress.bgn.ch4.basic;

import  java.lang.Math;

public class Sample extends Object {
    public static void main(String... args)  {
        System.out.println("PI value =" + Math.PI);

        double result = Math.sqrt(5.0);


        System.out.println("SQRT value =" + result);
    }
}
```

Another thing that you probably do when writing Java code is to **compact** import statements. Compacting imports is recommended when using multiple classes from the same package to write code, or multiple static variables and methods from the same class. When doing so, the import section of a file becomes really big and difficult to read. This is where compacting comes to help. Compacting imports means replacing all classes from the same package or variables and methods from the same class with a wildcard so only one import statement is needed. So, the Sample class becomes

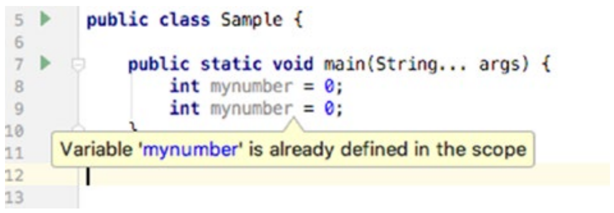
```
package com.apress.bgn.ch4.basic;

import static java.lang.Math.*;

public class Sample extends Object {
    public static void main(String... args) {
        System.out.println("PI value  =" + PI);

        double result = sqrt(5.0);

        System.out.println("SQRT value =" + result);
    }
}
```

Java “Grammar”

Java is case sensitive, which means that you can write a piece of code as follows.

```
public class Sample {

    public static void main(String... args) {
        int mynumber = 0;
        int myNumber = 1;
        int Mynumber = 2;
        int MYNUMBER = 3;
        System.out.println(mynumber);
        System.out.println(myNumber);
        System.out.println(Mynumber);
        System.out.println(MYNUMBER);
    }
}
```

All four variables are different and the last lines print numbers: `0 1 2 3`. You cannot declare two variables sharing the same name, in the same context (e.g., in the body of a method), because you would be basically redeclaring the same variable and the Java compiler does not allow this. If you try to do this, your code will not compile, and even IntelliJ IDEA will try to make you see the error of your ways by underlining the code in red and showing you a relevant message, like in Figure 4-1, where the `mynumber` variable is declared twice.



```
public class Sample {

    public static void main(String... args) {
        int mynumber = 0;
        int mynumber = 0;
    }
```

Variable 'mynumber' is already defined in the scope

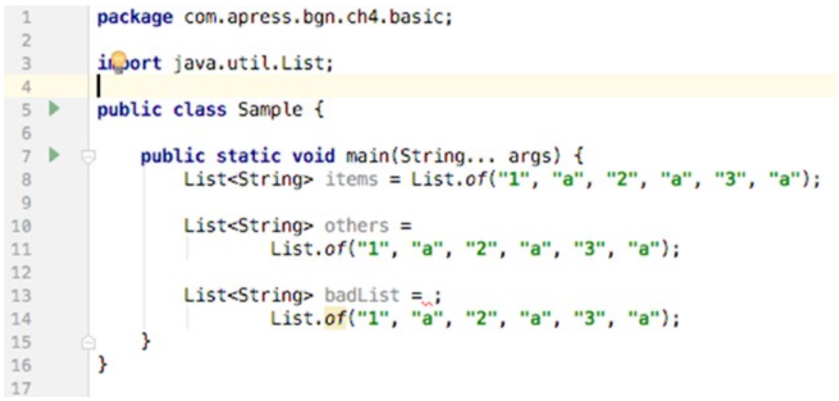
Figure 4-1. *Same statements example with error*

There is a set of **Java keywords** that can be used only for a fixed and predefined purpose in the Java code. A few of them have already been introduced: `import`, `package`, `public`, `class`. The rest of them are covered at the end of the chapter with a short explanation for each (see Tables 4-2 and 4-3).

Except for `import`, `package`, `interface` (or `@interface`), `enum` and `class` declarations, everything else in a Java source file must be declared between **curly brackets** (`{}`). These are called **block delimiters**. Take a look at the beginning of section 4.1. The brackets are used there to wrap up the following.

- contents of a class, also called the `body` of the class (brackets in lines 08 and 23)
- contents of a method, also called the `body` of a method (brackets in lines 09 and 22)
- a set of instructions to be executed together (brackets in lines 12 and 21)

Line terminators: code lines are usually ended in Java by the semicolon (;) symbol or by the ASCII characters CR, LF, or CR LF. Colons are used to terminate fully functioning statements, like the list declaration in line 11. If we have a really little monitor, and we are forced to split that statement on two subsequent lines to keep the code readable, the colon at its end tells the compiler that this statement that is correct only when taken together. Take a look at Figure 4-2.



```
1   package com.apress.bgn.ch4.basic;
2
3   import java.util.List;
4
5   public class Sample {
6
7       public static void main(String... args) {
8           List<String> items = List.of("1", "a", "2", "a", "3", "a");
9
10          List<String> others =
11                  List.of("1", "a", "2", "a", "3", "a");
12
13          List<String> badList =;
14                  List.of("1", "a", "2", "a", "3", "a");
15      }
16  }
17
```

Figure 4-2. *Different statements samples*

The declaration of a list in line 8 is equivalent to the one in lines 10 and 11. The declaration in line 13 and 14 is intentionally written wrong—a colon is added in line 13, which ends the statement there; but that statement is not valid and the compiler complains about it when you try to compile that class by printing an exception saying: `"Error:(13, 32) java: illegal start of expression"`. If the error message does not seem to fit the example, think about it like this: the problem for the compiler is not the wrongful termination of the statement, but that after the = symbol, the compiler expects to find some sort of expression that produces the value for the `badList` variable, but instead it finds nothing.

Java Identifiers

An **identifier** is the name you give to an item in Java: a class, variable, method, and so forth. Identifiers must respect a few rules to allow the code to compile and also common-sense programming rules, called **Java coding conventions**. A few of them are listed below:

- an identifier cannot be one of the Java reserved words, or the code will not compile
- an identifier cannot be a boolean literal (`true, false`) or the `null` literal , or the code will not compile
- an identifier can be made of letters, numbers and any of `_`, `$`
- developers should declare their identifiers following the Camel case writing style, the practice of writing compound words or phrases such that each word or abbreviation in the middle of the phrase begins with a capital letter, with no intervening spaces or punctuation, making sure each word or abbreviation in the middle of the identifier name begins with a capital letter (e.g., `StringBuilder`, `isAdult`)

A **variable** is a set of characters that can be associated with a value. It has a type. The set of values that can be assigned to it are restricted to a certain interval group of values or must follow a certain form defined by that type. For example, `items` declared in line 11 is a variable of type `List`.

In Java, there are three types of variables.

- **fields** are variables defined in class bodies, outside of method bodies and that do not have the keyword `static` in front of them
- **local variables** are variables declared inside method bodies, they are relevant only in that context
- **static variables** are variables declared inside class bodies with the have the keyword `static` in front of them. If they are declared as public they are accessible globally.

Java Comments

Java comments refer to pieces of explanatory text that are not part of the code executed and are ignored by the compiler. There are three ways to add comments within the code in Java, depending on the characters used to declare them.

- `//` is used for single line comments (line 10)
- `/** ... */` Javadoc comments, special comments that are exported using special tools into the documentation of a project called Javadoc API (lines 05 to 07)
- `/* ... */` used for multiline comments (lines 13 to 15)

Java Object Types

When introducing the Java building blocks in **Chapter** 3, only class was mentioned to keep things simple. It was mentioned that there are other object types in Java. The expression **object type** is not really accurate and in this section, things become clearer.

Classes are templates for creating objects. Creating an object based on a class is called **instantiation** and the resulted object is referred to as **an instance of that class**. Instances are called **objects** because by default any class written by a developer implicitly extends class `java.lang.Object` if no other superclass is declared. So, the following class declaration

```
package com.apress.bgn.ch4.basic;

public class Sample {
}
```

is equivalent to

```
package com.apress.bgn.ch4.basic;

public class Sample extends Object {
}
```

Also, notice how importing the `java.lang` package is not necessary, because the `Object` class is the root class of the Java hierarchy, all classes (including arrays) must have access to extend it. And thus, the `java.lang` package is implicitly imported as well.

But aside from classes, there are other template types that can be used for creating objects in Java. The following sections introduce them and explain what they are used for. But let's do so in context.

Let's create a family of templates for defining humans. Most Java tutorials use templates for vehicles or geometrical shapes. I want to model something that anybody can easily understand and relate to. The purpose of the following sections is to develop Java templates that model different types of people. The only Java template that I've explained so far is the class, so let's continue with that.

Classes

The operation through which instances are created is called **instantiation**. So, to design a class that models a generic human, we should think about two things: human characteristics and human actions. So, what do all humans have in common? Well, a lot, but for the purpose of this section, let's choose three generic attributes: a name, age, and height. These attributes map in a Java class to variables called **fields** or **properties**.

Fields

So, our class looks like this (initially):

```
package com.apress.bgn.ch4.basic;

public class Human {
    String name;

    int age;

    float height;
}
```

In the code sample, the **fields** have different types, depending on which values should be associated with them. For example, `name` can be associated with a text value, like "John", and text is represented in Java by the `String` type. The `age` can be associated with numeric integer values, so is of type `int`. And for the purpose of this section, we've considered that the height of a person is a rational number like 1.9, so we used the special Java type for this kind of value: `float`.

So, now we have a class modelling some basic attributes of a human. How do we use it? We need a `main()` method and we need to instantiate the class. In the next code snippet, a human named John is created.

```
package com.apress.bgn.ch4.basic;

public class BasicHumanDemo {

    public static void main(String... args) {
        Human human = new Human();
        human.name = "John";
        human.age = 40;
        human.height  =  1.91f;
    }
}
```

To create a `Human` instance, we use the `new` keyword. Next, we call a special method called a `constructor`. I've covered methods before, but this one is special. (Some programmers do not even consider it a method.) The most obvious reason for that is it wasn't defined anywhere in the body of the `Human` class. So, where is it coming from? Well, it's a default constructor that is automatically generated by the compiler unless an explicit one is declared. A class cannot exist without a constructor; otherwise, it cannot be instantiated. That is why the compiler generates one if none was explicitly declared. The default constructor, calls `super()` that invokes the `Object` no argument constructor that initializes all fields with default values. This can be tested by the following example.

```
package com.apress.bgn.ch4.basic;

public class BasicHumanDemo {

    public static void main(String... args) {
        Human human = new Human();
        System.out.println("name: " + human.name);
        System.out.println("age:  " +  human.age);
        System.out.println("height: " + human.height);
    }
}
```

What do you think will happen when you run the previous code? If you think that some default values (neutral) printed, you are absolutely right. The following is the output of the previous code.

```
name: null
age: 0
height: 0.0
```

The numeric variables were initialized with 0, and the String value was initialized with null. The reason for that is that the numeric types are primitive data types and String is an object data type. The String class is part of the java.lang package, which is one of the predefined Java classes that creates objects of type String. It is a special data type that represents text objects. We'll go deeper into data types in the following chapter.

Class Variables

Aside attributes that are specific to each human in particular, all humans have something in common: a lifespan, which is assumed to be 100 years. It would be redundant to declare a field called *lifespan*, because it has to be associated with the same value for all human instances. So, we declare a field using the static keyword in the Human class, which has the same value for all Human instances and that is initialized only once. And we can go one step further and make sure that value never changes during the execution of the program by adding the final modifier in front of its declaration as well. This way we created a special type of variable called a **constant**. The new Human class looks like this:

```
package com.apress.bgn.ch4.basic;

public class Human {
    static final int LIFESPAN = 100;

    String name;

    int age;

    float height;

}
```

The `LIFESPAN` variable is also called a **class variable**, because it is not associated with instances but with the class. This is clear in the following example.

```
package com.apress.bgn.ch4.basic;

public class BasicHumanDemo {

    public static void main(String... args) {
        Human john = new Human();
        john.name = "John";

        Human jane = new Human();
        jane.name = "Jane";

        System.out.println("John’s lifespan = " + john.LIFESPAN);
        System.out.println("Jane’s lifespan = " + jane.LIFESPAN);

        System.out.println("Human lifespan = " + Human.LIFESPAN);
    }
}
```

When the `main()` method of the preceding class is executed, the following is printed, which proves everything that was mentioned before.

```
John's lifespan = 100
Jane's lifespan = 100
Human lifespan = 100
```

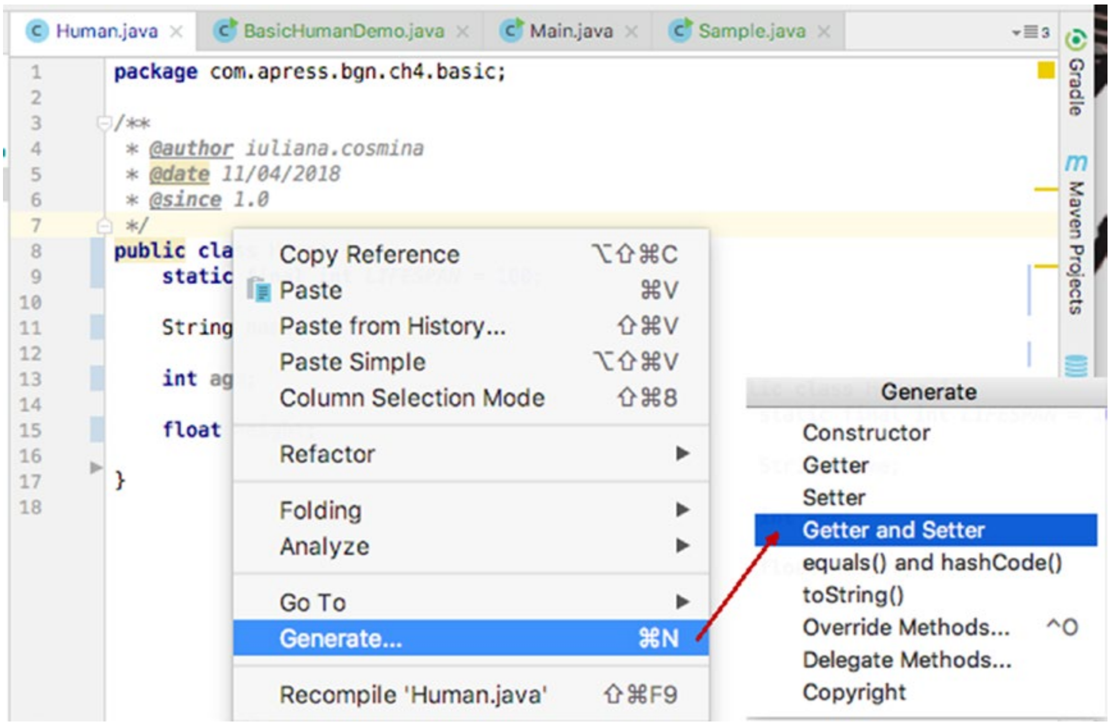
Encapsulating Data

The class we defined makes no use of access modifiers on the fields, which is not acceptable. Java is known as an object-oriented programming language (OOP), and thus, code written in Java must respect **the principles of OOP**. Respecting these coding principles ensures that the written code is of good quality and totally aligns with the fundamental Java style. One of the OOP principles is **encapsulation**. The encapsulation principle refers to hiding of data implementation by restricting access to it using special methods called **accessors (getters)** and **mutators (setters)**.

Basically, any field of a class should have private access, and access to it should be controlled by methods that can be intercepted, tested, and tracked to see where they were called. Getters and setters are a normal practice to have when working so objects that most IDEs have a default options to generate them, including IntelliJ IDEA. Right-click inside the class body and select the **Generate** option to see all possibilities and select **Getters and Setters** to generate the methods for you. The menu is depicted in Figure 4-3.

After making the fields private, and generating the getters and setter the `Human` class now looks like this:

```
package com.apress.bgn.ch4.basic;

public class Human {
    static final int LIFESPAN = 100;

    private String name;

    private int age;

    private float height;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public float getHeight() {
        return height;
    }
    public void setHeight(float height) {
        this.height = height;
    }
}
```



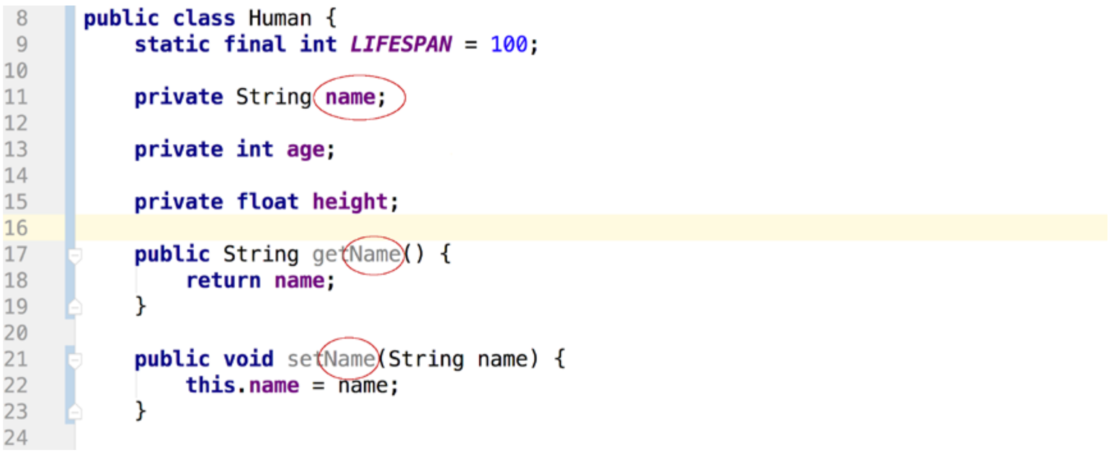
```
package com.apress.bgn.ch4.basic;

/**
 * @author iuliana.cosmina
 * @date 11/04/2018
 * @since 1.0
 */
public cla
    static

    String

    int ag

    float
}
```

Figure 4-3. *IntelliJ IDEA code generation menu. Generate... ➤ Getter and Setter submenu*

So, you may be wondering what `this` is. As the word says, it is a reference to the current object. So, `this.name` is the value of the field `name` of the current object. Inside the class body, `this` accesses fields for the current object, when there are parameters in methods that have the same name. And as you can see, the setters and getters that IntelliJ IDEA generates have parameters that are named the same as the fields.

Getters are the simplest methods declared without any **parameter**. They return the value of the field they are associated with. Their naming convention uses the `get` prefix and the name of the field they access, with the first letter uppercased.

Setters are methods that return nothing. They declare as a parameter a variable with the same type that needs to be associated to the field. Their names are made of the `set` prefix and the name of the field they access, with its first letter uppercased. Figure 4-4 depicts the setter and getter for the **name** field.



```java
public class Human {
    static final int LIFESPAN = 100;

    private String name;

    private int age;

    private float height;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

```

Figure 4-4. *Setter and getter methods used for the name field*

This means that when instantiating the `Human` class, we have to use the setters to set the field values and the getters to access those values. Thus, our class `BasicHumanDemo` becomes

```java
package  com.apress.bgn.ch4.basic;

public class BasicHumanDemo {

    public static void main(String... args) {
        Human human = new Human();
        human.setName("John");
        human.setAge(40);
        human.setHeight(1.91f);

        System.out.println("name: " + human.getName());
        System.out.println("age: " + human.getAge());
        System.out.println("height: " + human.getHeight());
    }
```

Methods

Since getters and setters are methods it is time to start the discussions about methods too. A method is a block of code characterized by returned type, name, and parameters that describes an action done by or on the object that makes use of the values of its fields and/or arguments provided. An abstract template of a Java method is depicted as follows.

```
[accessor] [returned type] [name] type1 param1, type2 param2, ... {
 // code
 [ [maybe] return val]
}
```

Let's create a method for the `Human` class that computes and prints how much time a human still has to live by making use of his age and the `LIFESPAN` constant. Because the method does not return anything, the return type used is `void`, a special type that tells the compiler that the method does not return anything and we have no return statement in the method body.

```
package com.apress.bgn.ch4.basic;

public class Human {
    static final int LIFESPAN = 100;

    private String name;
    private int age;
    private float height;

     /**
     * compute and prints time to live
     */
    public void computeAndPrintTtl(){
        int ttl = LIFESPAN - this.age;
        System.out.println("Time to live: " + ttl);
    }
        ...
}
```

! There is a Java coding convention in the naming of constants that recommends using only uppercase letters, underscores, and numbers.

The preceding method definition does not declare any parameters, so considering we have a Human instance we can call the method like this:

```
Human human = new Human();
human.setName("John");
human.setAge(40);
human.setHeight(1.91f);
human.computeAndPrintTtl();
```

And we expect it to print `Time to live: 60`, which actually happened. Now, let's modify the method to return the value instead of printing it.

```
package com.apress.bgn.ch4.basic;

public class Human {
    static final int LIFESPAN = 100;

    private String name;
    private int age;
    private float height;

    /**
     * @return time to live
     */
    public int getTimeToLive(){
        int ttl = LIFESPAN - this.age;
        return ttl;
    }
        ...
}
```

Calling the method do nothing in this case, we have to modify the code to save the returned value and print it.

```
Human human = new Human();
human.setName("John");
human.setAge(40);
human.setHeight(1.91f);
int timeToLive = getTimeToLive();
System.out.println("Time  to  live: " + timeToLive);
```

Both methods introduced here declare no parameters, so they are called without providing any arguments. We won't cover methods with parameters, as the setters are more than obvious. Let's skip ahead.

Constructors

Now we've done it. We can no longer use `human.name` without the compiler complaining about it. But still, it is annoying to call all of those setters to set the properties; something should be done about that. Remember the implicit constructor? Well, let's create an explicit one that has parameters for each of the fields we are interested in.

```
public class Human {
    static final int LIFESPAN = 100;

    private String name;
    private int age;
    private float height;

    public Human(String name, int age, float height) {
        this.name = name;
        this.age =  age;
        this.height   =   height;
    }
    ...
}
```

In the preceding example, you can see that the constructor does not include a `return` statement, even if the result of calling a constructor is the creation of an object. Constructors are different from methods in that way. By declaring an explicit constructor, the default constructor is no longer generated. So, creating a `Human` instance by calling the default constructor does not work anymore; the code no longer compiles because the default constructor is no longer generated.

```
Human human = new Human();
```

To create a `Human` instance, we now have to call the new constructor and provide proper arguments in place of the parameters, having the same types as declared.

```
Human human = new Human("John", 40, 1.91f);
```

But what if we do not want to be forced to set all fields using this constructor? It's simple, we define another with only the parameters that we are interested in. Let's define a constructor that only sets the name and the age for a `Human` instance.

```
public class Human {
    static final int LIFESPAN = 100;

    private String name;
    private int age;
    private float height;

    public Human(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Human(String name, int age, float height) {
        this.name = name;
        this.age = age;
        this.height = height;
    }
    ...
}
```

And this is where we stumble upon an OOP principle called **polymorphism**. The term is Greek and translates to *one name, many forms*. Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality. There are two basic types of polymorphism: **overriding**, also called **run-time polymorphism**, and **overloading**, which is referred to as **compile-time polymorphism**. The second type of polymorphism applies to the preceding constructors, because we have two of them, one with a different set of parameters that looks like it is an extension of the simpler one.

So, we have some code duplication in the previous example, and there is a common sense programming principle called **DRY**[1] (Don't Repeat Yourself!) that the following example clearly defies. So, let's fix that by using the `this` keyword.

```
public class Human {
    static final int LIFESPAN = 100;

    private  String  name;
    private int age;
    private float height;

    public Human(String name, int age) {
        this.name = name;
        this.age  =  age;
    }

    public Human(String name, int age, float height) {
        this(name, age);
        this.height  =  height;
    }
    ...
}
```

Yes, constructors can call each other by using `this(...)`. So now, we can use both constructors to create `Human` instances. If we use the one that does not set the height, the `height` field is implicitly initialized with the default value for type `float`.

[1]Also one of the clean coding principles; read more about it at https://blog.goyello.com/2013/01/21/top-9-principles-clean-code/

Now, our class is generic; we could even say that it models a Human class in an abstract way. If we were to try to model humans with certain skill sets or abilities, we must enrich this class. Let's say we want to model musicians and actors. This means we need to create two new classes. The Musician class is depicted in the following; getters and setters for the fields are skipped.

```
public class Musician {
   static final int LIFESPAN = 100;

   private String name;

   private int age;

   private float height;

   private String musicSchool;

   private  String  genre;

   private List<String> songs;
   ...
}
```

The Actor class is depicted next; getters and setters for the fields are also skipped.

```
public class Actor {
    static final int LIFESPAN = 100;

    private String name;

    private int age;

    private float height;

    private  String  actingSchool;

    private List<String> films;
    ...
}
```

There are more than a few common elements between the two classes. One of the clean coding principles requires developers to avoid code redundancy. This can be done by designing the classes by following two OOP principles: **inheritance** and **abstraction**.

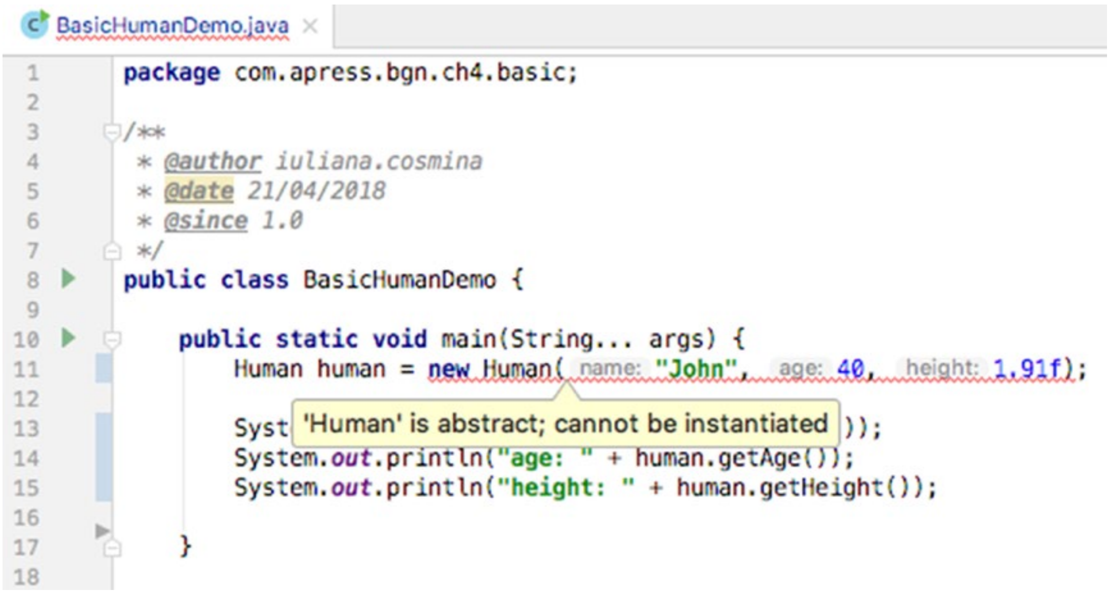
Abstraction

Abstraction is an OOP principle that manages complexity. Abstraction decomposes complex implementations and defines core parts that can be reused. In our case, common fields of the Musician and Actor classes can be grouped in the Human class that we defined earlier in the chapter. The Human class can be viewed as an abstraction, because any human in this world is more than his name, age, and height. So, there is no need to create Human instances, because a human is represented by something else, like passion, purpose, and skill. A class that does not need to be instantiated, but groups together fields and methods for other classes to inherit, or provide a concrete implementation for is modelled in Java by an abstract class. Thus, we modify the Human class to make it abstract first. And since we are abstracting this class, let's make the LIFESPAN constant public so we can access it from anywhere and make the getTimeToLive method abstract.

```
package com.apress.bgn.ch4.basic;

public abstract class Human {
    public static final int LIFESPAN = 100;

    private String name;
    private int age;
    private float height;

    public Human(String name, int age) {
        this.name = name;
        this.age  =  age;
    }

    public Human(String name, int age, float height) {
        this(name, age);
        this.height  =  height;
    }

    /**
     * @return time to live
     */
    public abstract int getTimeToLive();
...
// setters & getters for fields in this class
}
```

An abstract method like `getTimeToLive()` is declared in the example; it is a method missing the body. This means that within the `Human` class, there is no concrete implementation for this method, only a skeleton—a template that extending classes must provide a concrete implementation for.

Oh, but wait, we kept the constructors! Why did we do that if we are not allowed to use them anymore? And we aren't, because Figure 4-5 shows what IntelliJ IDEA does with the `BasicHumanDemo` class Figure 4-5.


BasicHumanDemo.java

```
package com.apress.bgn.ch4.basic;

/**
 * @author iuliana.cosmina
 * @date 21/04/2018
 * @since 1.0
 */
public class BasicHumanDemo {

    public static void main(String... args) {
        Human human = new Human( name: "John",  age: 40,  height: 1.91f);

        Syst 'Human' is abstract; cannot be instantiated ));
        System.out.println("age: " + human.getAge());
        System.out.println("height: " + human.getHeight());

    }

```

Figure 4-5. *Java compiler error when trying to instantiate an abstract class*

We kept the constructors because they can help further abstracting behavior. The `Musician` and `Actor` classes must be rewritten to extend the `Human` class. This is done by using the `extends` keyword when declaring the class and specifying the class to be extended, also called the **parent class** or **superclass**. The resulting class is called a **subclass**. When extending a non-abstract class, the subclass **inherits** all the fields and concrete methods declared in the superclass.

When extending an abstract class, the subclass must provide a concrete implementation for all abstract methods, and must declare their own constructors, which eventually make use of the constructors declared in the abstract class. These constructors can be called by using the keyword `super`. The same goes for methods, but not for fields, unless they have the proper access modifier.

Let's see what the Musician class looks like when making use of abstraction and inheritance.

```
 package com.apress.bgn.ch4.basic;

import java.util.List;

public class Musician extends Human {

    private String musicSchool;

    private String genre;

    private List<String> songs;

    public Musician(String name, int age, float height,
        String musicSchool, String genre) {
        super(name, age, height);
        this.musicSchool = musicSchool;
        this.genre = genre;
    }

    public int getTimeToLive() {
        return (LIFESPAN - getAge()) / 2;
    }
...
// setters & getters for fields in this class
}
```

The songs field was not used as a parameter in the constructor for simplicity reasons here.

The Musician constructor calls the constructor in the superclass to set the properties defined there. Also, notice the full implementation provided for the getTimeToLive() method.

The Actor class is rewritten in a similar manner. You find a proposal implementation in the sources for the book, but try to write your own before looking in the com.apress.bgn.ch4.basic package.

Figure 4-6 shows the Human class hierarchy, as generated by IntelliJ IDEA.

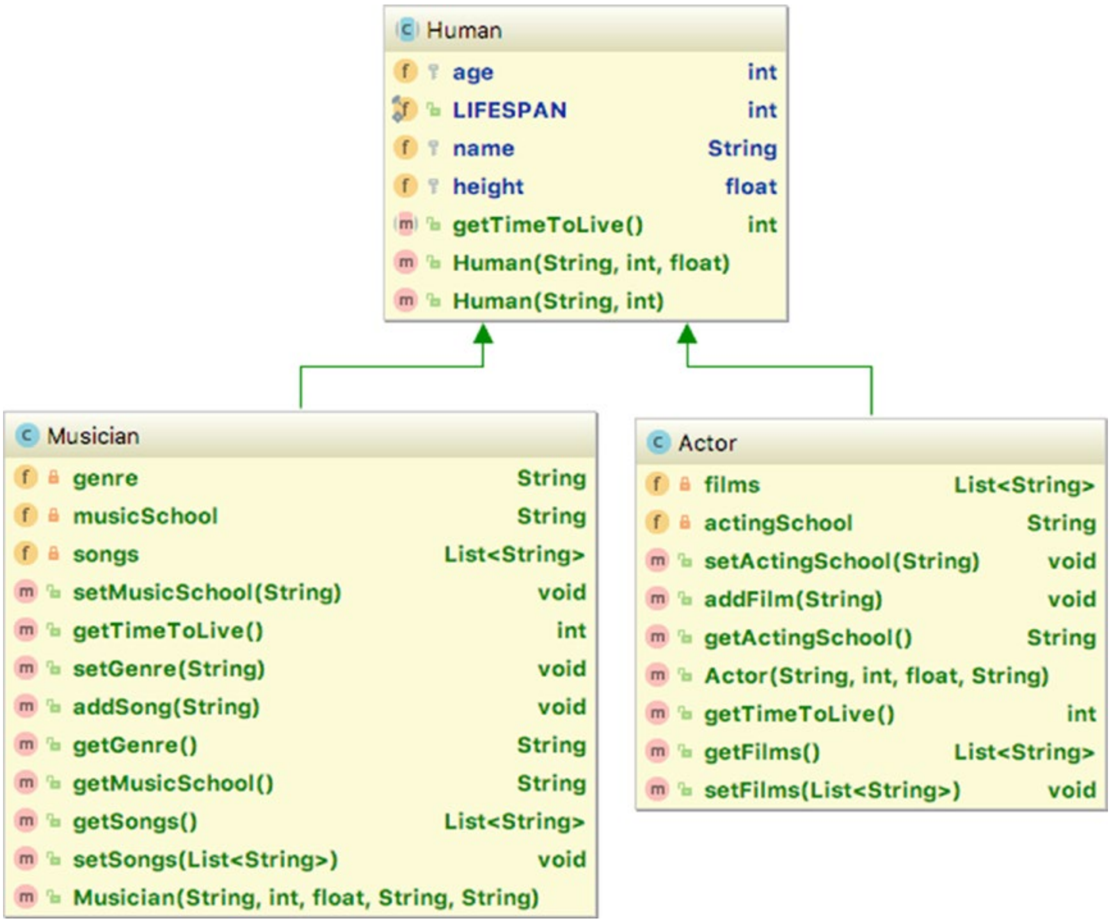


Figure 4-6. *UML diagram generated by IntelliJ IDEA*

The UML diagram clearly shows the members of each class and the arrows point to the superclass. UML diagrams are useful tools in designing class hierarchies and defining logic of applications. If you want to read more about them and the many types of UML diagrams that there are, you can do so at `www.uml-diagrams.org`.

After covering so much about classes and how to create objects, we need to cover other Java important components that create even more detailed objects, which can then be used to implement more complex applications. Our `Human` class is missing quite a few attributes, like `gender` for example. A field that models the gender of a person can only have values from a fixed set of values. It used to be two, but because we are living in a brave new world that is fond of political correctness, we cannot limit the set of values for genders to two; so we introduce a third, called *UNDEFINED*. This means that we must introduce a new class to represent a gender that is limited to being instantiated three times. This would be tricky to do with a typical class. So, in Java version 1.5, **enums** were introduced.

Enums

The enum type is a special class type. It defines a special type of class that can only be instantiated a fixed number of times. An enum declaration, groups all instances of that enum. All of them are constants. So, the Gender enum can be defined as shown in the following piece of code.

```
package com.apress.bgn.ch4.basic;

public enum Gender {
    FEMALE,
    MALE,
    UNDEFINED
}
```

An enum cannot be instantiated externally. An enum is by default final, thus it cannot be extended. Remember how by default every class in Java implicitly extends class Object? Every enum in Java implicitly extends class java.lang.Enum<E> and in doing so, every enum instance inherits special methods that are useful when handling enums.

As an enum is a special type of class, it can have fields and a constructor that can only be private, as enum instances cannot be created externally. The private modifier is not needed explicitly, as the compiler knows what to do. Let's modify our Gender enum to add an integer field that is the numerical representation of each gender and a String field that is the text representation.

```
package com.apress.bgn.ch4.basic;

public enum Gender {
    FEMALE(1, "f"),
    MALE(2, "m") ,
    UNDEFINED(3, "u");

    private int repr;
    private String descr;
    Gender(int repr, String descr) {
        this.repr = repr;
        this.descr = descr;
    }

    public int getRepr() {
        return repr;
    }

    public String getDescr() {
        return descr;
    }
}
```

But wait, what would stop us from declaring setters and modifying the field values? Well, nothing. If that is what you need to do you can do it. But **this is not a good practice**. Enum instances, should be constant. So, what we can do is to not create setters, and make sure the values of the fields never change by declaring them `final`. When we do so, the only way the fields can be initialized is by calling the constructor, and since the constructor cannot be called externally, the integrity of our data is ensured. So, our `enum` becomes

```
package com.apress.bgn.ch4.basic;

public enum Gender {
    FEMALE(1, "f"),
    MALE(2, "m") ,
    UNDEFINED(3, "u");

    private final int repr;
    private final String descr;

    Gender(int repr, String descr) {
        this.repr = repr;
        this.descr = descr;
    }

    public int getRepr() {
        return repr;
    }
    public String getDescr() {
        return descr;
    }
}
```

Methods can be added to enums, and each instance can override them. So, if we add a method called `getComment()` to the `Gender` enum, every instance inherits it. But the instance can override it. Let's see what that looks like.

```
package com.apress.bgn.ch4.basic;

public enum Gender {
    FEMALE(1, "f"),
    MALE(2, "m") ,
    UNDEFINED(3, "u"){
        @Override
        public String comment() {
           return "to be decided later: " + getRepr() + ", " + getDescr();
        }
    };

    private final int repr;
    private final String descr;

    Gender(int repr, String descr) {
        this.repr = repr;
        this.descr = descr;
    }

    public int getRepr() {
        return repr;
    }

    public String getDescr() {
        return descr;
    }

    public String comment() {
        return repr + ": " + descr;
    }
}
```

If we were to print the values returned by the `comment()` method for each instance, we would see the following.

```
package com.apress.bgn.ch4.basic;

public class Sample extends Object {
    public static void main(String... args) {
        System.out.println(Gender.FEMALE.comment());
        // prints '1: f'
        System.out.println(Gender.MALE.comment());
        // prints '2: m'
        System.out.println(Gender.UNDEFINED.comment());
        //prints 'to be decided later: 3, u'
    }
}
```

We're going to be playing with enums in future examples as well. Just remember that whenever you need to limit the implementation of a class to a fixed number of instances, `enums` are the tools for you. And now because we introduced enums, our `Human` class can also have a field of type `Gender`.

```
package com.apress.bgn.ch4.basic;

public abstract class Human {
    public static final int LIFESPAN = 100;

    protected String name;

    protected int age;

    protected float height;

    private Gender gender;

    public Human(String name, int age, Gender gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
    public Human(String name, int age, float height, Gender gender) {
        this(name, age, gender);
        this.height = height;
    }
    ...
}
```

In previous sections, **interfaces** were mentioned as one of the Java tools used to create objects. It is high time I expand the subject.

Interfaces

One of the most common Java interview questions is, "What is the difference between an interface and an abstract class?" This section provides you the most detailed answer to that question. An **interface** is not a class, but it does help create classes. An interface is fully abstract; it has no fields, only method definitions (skeletons). A class can implement an interface, and unless the class is abstract, it is forced to provide concrete implementations for them. Each method declared inside an interface is implicitly public and abstract, because methods need to be abstract to force implementing classes to provide implementations and are public, so classes have access to do so.

The only methods with concrete bodies in an interface are static methods and starting with Java 8, **default** methods. The interfaces cannot be instantiated, they do not have constructors.

Interfaces that declare no method definitions are called **marker** interfaces and have the purpose to mark classes for specific purposes. The most renowned Java marker interface is `java.io.Serializable`, which marks objects that can be serialized(their state can be saved to a binary file).

An interface can be declared in its own file as a top-level component, or nested inside another component. There are two types of interfaces: normal interfaces and annotations.

The difference between abstract classes and interfaces, and when one or the other should be used, becomes relevant in the context of **inheritance**. Java supports only single inheritance. This means a class can only have one superclass. This might seem like a limitation, but let's consider a simple example. Let's modify the previous hierarchy and imagine a class called `Performer` that should extend the `Musician` and `Actor` classes. If you need a real human that can be modelled by this class, think of David Duchovny, an actor who recently got into music.

Figure 4-7 shows the class hierarchy.

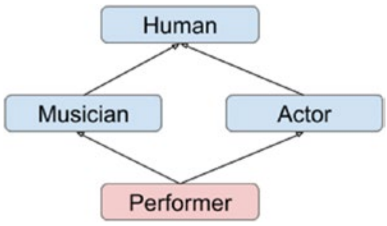


Figure 4-7. *Diamond class hierarchy*

The hierarchy in Figure 4-7 introduces something called **the diamond problem**, and the name is inspired by the shape formed by the relationships between classes. What is actually wrong with the design? If both `Musician` and `Actor` extend `Human`, and inherit all members from it, which member does `Performer` inherit and from where? Because it cannot inherit members of the `Human` class twice - this would make this class useless and invalid. So, what is the solution? As you probably imagine, given the title of this section: **interfaces**.

What has to be done is to turn methods in classes `Musician` and `Actor` into method skeletons and transform those classes into interfaces. The behavior from the `Musician` is moved to a class called, let's say `Guitarist`, which extends the `Human` class and implement the `Musician` interface. For the `Actor` class, something similar can be done, but I'll leave that as an exercise for you. Some help is provided by the hierarchy shown in Figure 4-8.

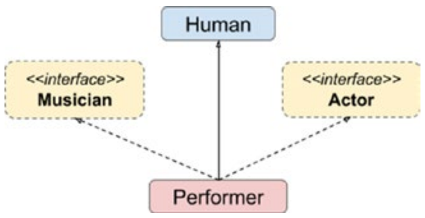


Figure 4-8. *Java hierarchy with interfaces for Performer class*

The `Musician` interface contains only method templates mapping what a musician does. It does not go into detail to model how. The same goes for the `Actor` interface. In the following code snippet, you can see the bodies of the two interfaces.

```
 // Musician.java
 package com.apress.bgn.ch4.hierarchy;

import java.util.List;
```

```
public interface Musician {
    String getMusicSchool();
    void setMusicSchool(String musicSchool);
    List<String> getSongs();
    void setSongs(List<String> songs);
    String getGenre();
    void setGenre(String genre);
}

 // Actor
package com.apress.bgn.ch4.hierarchy;

import java.util.List;

public interface Actor {
    String getActingSchool();
    void setActingSchool(String actingSchool);
    List<String> getFilms();
    void setFilms(List<String> films);
    void addFilm(String filmName);
}
```

The fields have been removed because they cannot be part of the interfaces; all that is left are the method templates. The `Performer` class is depicted in the next code snippet.

```
 package com.apress.bgn.ch4.hierarchy;

import java.util.List;

public class Performer extends Human
    implements Musician, Actor {

    private String musicSchool;

    private String genre;

    private List<String> songs;

    private String actingSchool;

    private List<String> films;
    public Performer(String name, int age, float height, Gender gender) {
        super(name, age, height, gender);
    }

    @Override
    public int getTimeToLive() {
        return (LIFESPAN - getAge()) / 2;
    }

    public String getMusicSchool() {
        return musicSchool;
    }

    public void setMusicSchool(String musicSchool) {
        this.musicSchool = musicSchool;
    }

    public List<String> getSongs() {
        return songs;
    }

    public void setSongs(List<String> songs) {
        this.songs = songs;
    }

    public void addSong(String song) {
        this.songs.add(song);
    }

    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
        this.genre = genre;
    }

    public String getActingSchool() {
        return actingSchool;
    }
    public void setActingSchool(String actingSchool) {
        this.actingSchool = actingSchool;
    }

    public List<String> getFilms() {
        return films;
    }

    public void setFilms(List<String> films) {
        this.films = films;
    }

    public void addFilm(String filmName) {
        this.films.add(filmName);
    }
}
```

What you are left with from this example is that using interfaces **multiple inheritance** is possible in Java, and that classes extend classes and implement interfaces. But inheritance applies to interfaces too. For example, both Musician and Actor interface can extend an interface named Artist that contains template for behavior common to both. For example, we can combine the music school and acting school into a generic school and define the setters and getters for it as method templates. The Artist interface is depicted as follows with Musician.

```
 // Artist.java
package com.apress.bgn.ch4.hierarchy;

public interface Artist {
    String getSchool();
    void setSchool(String chool);
}

 // Musician.java
package com.apress.bgn.ch4.hierarchy;

import java.util.List;
```

```
public interface Musician extends Artist {

    List<String> getSongs();
    void setSongs(List<String> songs);
    String getGenre();
    void setGenre(String genre);
}
```

Hopefully, you understood the idea of multiple inheritance, when it is appropriate to use classes, and when to use interfaces in designing your applications. It is time to fulfill the promise made in the beginning of this section and list the differences between abstract classes and interfaces. You can find them in Table 4-1.

Table 4-1. *Differences Between Abstract Classes and Interfaces in Java*

Abstract Class	**Interface**
Can have non-abstract methods	Can only have abstract and (since Java 8 default methods, since Java 9 private methods)
Single inheritance: a class can only extend one class	Multiple inheritance: a class can implement more than one interface.
Can have final, non-final, static and non-static variables	Can only have static and final fields.
Declared with **abstract class**	Declared with **interface**.
Can extend another class using keyword **extends** and implement interfaces with keyword **implements**	Can only extend other interfaces (one or more) using key-word **extends**.
Can have non-abstract, protected or private members	All members are method definitions and are by default abstract and public. (Except default methods, starting with Java 8 and private methods, starting with Java 9.)
If a class has an abstract method, it must be declared itself abstract	(No correspondence)

Default Methods

One problem with interfaces is that if you modify their bodies to add new methods, most likely, the code stops compiling because the classes implementing the interfaces do not provide concrete implementations for the new methods declared in the interfaces. Sure, a solution would be to declare the new methods in a new interface and then creating new classes that implement both new and old interfaces.

The methods interfaces expose make up an API (application programming interface) and when developing applications, the aim is to design applications and their components to have a stable API. This rule is described in the **open closed principle**, which is one of the five SOLID programming principles.[2] This principle states that you should be able to extend a class without modifying it. Thus, modifying the interface a class implements, extends the class behavior, but only if the class is modified to provide a concrete implementation for the new methods. So, implementing interfaces, tends to lead to breaking this principle. So, how can we avoid this in Java?

In Java 8, a solution for this was finally introduced: **default methods**. Starting with Java 8, methods with a full implementation can be declared in interfaces as long as they are declared using the `default` keyword.

Let's consider the previous example: the `Artist` interface. Any artist should be able to create something, right? So, he or she should have a creative nature. Given the world we are living in, I won't mention names, but some of our artists are actually products of the industry, so they are not creative themselves. So, the realization that we should have a method that tells us if an artist has a creative nature or not, came way after we decided our hierarchy, which is depicted in Figure 4-9.

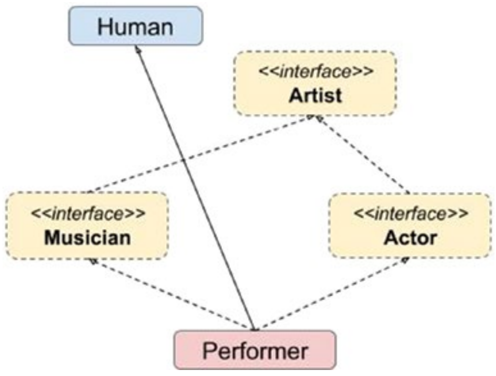
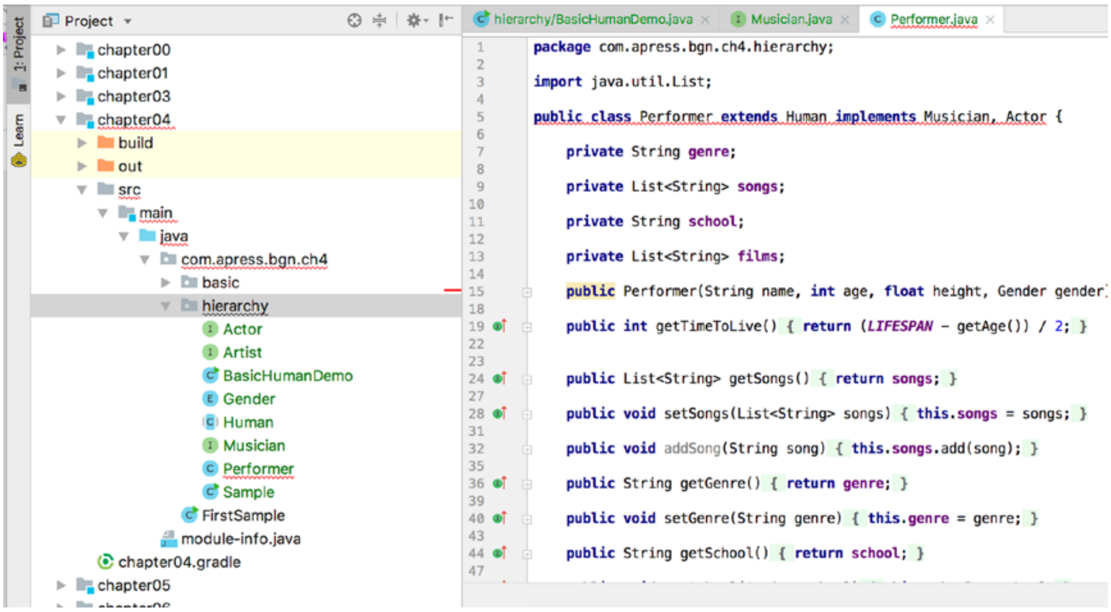


Figure 4-9. *Java hierarchy with more interfaces for Performer class*

[2]A good article is at `https://hackernoon.com/solid-principles-made-easy-67b1246bcdf`

If we add a new method template to the `Artist` interface, the `Performer` class causes a compile error. IntelliJ IDEA makes it clear that our application does not work anymore by showing a lot of things in red, as depicted in Figure 4-10.



```
chapter00
chapter01
chapter03
chapter04
    build
    out
    src
        main
            java
                com.apress.bgn.ch4
                    basic
                    hierarchy
                        Actor
                        Artist
                        BasicHumanDemo
                        Gender
                        Human
                        Musician
                        Performer
                        Sample
                    FirstSample
                module-info.java
    chapter04.gradle
chapter05
```

```java
package com.apress.bgn.ch4.hierarchy;

import java.util.List;

public class Performer extends Human implements Musician, Actor {

    private String genre;

    private List<String> songs;

    private String school;

    private List<String> films;

    public Performer(String name, int age, float height, Gender gender
    public int getTimeToLive() { return (LIFESPAN - getAge()) / 2; }


    public List<String> getSongs() { return songs; }

    public void setSongs(List<String> songs) { this.songs = songs; }

    public void addSong(String song) { this.songs.add(song); }

    public String getGenre() { return genre; }

    public void setGenre(String genre) { this.genre = genre; }

    public String getSchool() { return school; }
```

Figure 4-10. *Java broken hierarchy*

The compiler errors that we see are caused by our decision to add a new method, named `isCreative`, to the `Artist` interface. It is underlined in the following code snippet.

```java
package com.apress.bgn.ch4.hierarchy;

public interface Artist {
    String getSchool();

    void setSchool(String school);


    boolean isCreative();
}
```

To get rid of the compiling errors we'll transform the `isCreative` method into a default method that returns `true`, because every artist should be creative.

```
package com.apress.bgn.ch4.hierarchy;

public interface Artist {
    String getSchool();

    void setSchool(String school);


    default boolean isCreative(){
        return true;
    }
}
```

Now, the code should compile again. If we need to add more than one default method to an interface and the methods have some implementation in common, that code can be isolated starting with Java 9 into a private method that can be called from the default methods. So basically, starting from Java 9, full blown methods can be part of an interface, as long as they are declared private.

Annotation Types

An annotation is defined in a similar way to an interface; the difference is that the `interface` keyword is preceded by the *at* sign (`@`). Annotation types are a form of interfaces, and most times, they are used as markers. For example, you've probably noticed the `@Override` annotation. This annotation is automatically placed by intelligent IDEs when classes extending or implementing interfaces are generated automatically. It's declaration in the JDK is depicted in the following code snippet.

```
package java.lang;

import  java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {}
```

Annotations that do not declare any property are called **marker** or **informative** annotations. They are needed only to inform other classes in the application, or developers of the purpose of the components they are placed on. They are not mandatory and the code compiles without them.

In Java 8, an annotation named @FunctionalInterface was introduced. This annotation was placed on all Java interfaces that can be used in **lambda expressions**.

```
 package java.lang;

import java.lang.annotation.*;
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

Lambda expressions were also introduced in Java 8 and they represent a compact and practical way of writing code that was borrowed from languages like Groovy and Ruby.

Functional Interfaces are interfaces that declare a single abstract method. Because of this, the implementation of that method can be provided on the spot, without the need to create a class to define a concrete implementation.

Let's imagine the following scenario: we create an interface named Operation that contains a single method.

```
package com.apress.bgn.ch4.lambda;

@FunctionalInterface
public interface Operation {
    float execute(int a, int b);
}
```

We'll next create a class named Addition.

```
package com.apress.bgn.ch4.lambda;

public class Addition implements Operation {

    @Override
    public float execute(int a, int b) {
        return a + b;
    }
}
```

And if we want to test it, we need yet another class.

```
package com.apress.bgn.ch4.lambda;

public class OperationDemo {
    public static void main(String... args) {
        Addition addition = new Addition();
        float  result = addition.execute(2,5);
        System.out.println("Result is " + result);
    }
}
```

Using lambda, the `Addition` class is no longer needed, and the instantiation and the method call can be replaced with

```
package com.apress.bgn.ch4.lambda;

public class OperationDemo {
    public static void main(String... args) {
        Operation addition2 = (a, b) -> a + b;
        float result2 = addition2.execute(2, 5);
        System.out.println("Lambda Result is " + result2);
    }
}
```

Lambda expressions can be used for a lot of things. I'll cover them throughout the book, whenever code can be written in a more practical way using them.

Exceptions

Exceptions are special Java classes that are used to intercept special unexpected situations during the execution of a program so that the developer can implement the proper course of action. These classes are organized in a hierarchy that is depicted in Figure 4-11.

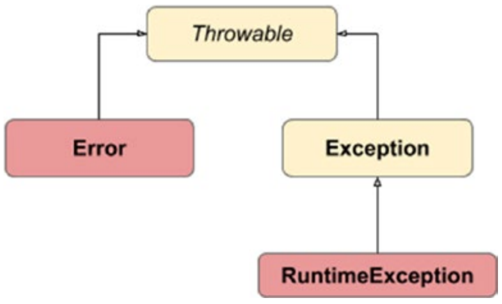


Figure 4-11. *Java Exception hierarchy*

`Throwable` is the superclass of all errors that can be thrown in a Java application. The exceptional situations can be caused by hardware failures (e.g., trying to read a protected file), by missing resources (e.g., trying to read a file that does not exist), or by bad code. Bad developers tend to do this: when in doubt, catch a throwable. You should definitely try to avoid this because the `Error` class that notifies the developer about a situation that the system cannot recover from is a subclass of it. Let's start with a simple example. We define a method that calls itself (its technical name is **recursive**), but we'll design it badly to call itself forever and cause the JVM to run out of memory.

```
package com.apress.bgn.ch4.ex;

public class ExceptionsDemo {

    // bad method
    static int rec(int i){
        return rec(i*i);
    }

    public static void main(String... args) {
          rec(1000);
        System.out.println("An error happened.");
    }
}
```

If we run the class, *An error happened* is not printed. Instead, the program ends abnormally by throwing a `StackOverFlowError` and states the line where the problem is (in our case, the line where the recursive method calls itself).

```
Exception in thread "main" java.lang.StackOverflowError
        at chapter.four/com.apress.bgn.ch4.ex.ExceptionsDemo.
        recExceptionsDemo.java:7
        at chapter.four/com.apress.bgn.ch4.ex.ExceptionsDemo.
        recExceptionsDemo.java:7
        ...
```

StackOverFlowError is a subclass of Error, and is caused by the defective recursive method that was called. Sure, we could modify the code, treat this exceptional situation, and execute whatever has to be executed next.

```
package com.apress.bgn.ch4.ex;

public class ExceptionsDemo {
...
    public static void main(String... args) {
        try {
            rec(1000);
        } catch (Throwable r) {
        }
        System.out.println("An error happened.");
    }
}
```

In the console, you see only the *An error happened* text, but no trace of the error, which is why we caught it and decided not to print any information about it. This is also a bad practice called **exception swallowing**, never do this! Also, the system should not recover from this, as the result of any operation after an error is thrown is unreliable. That is why, as a rule of thumb, **never catch a throwable!!**

The Exception class is the superclass of all exceptions that can be caught and treated, and the system can recover from them. The RuntimeException class is the superclass of exceptions that are thrown during the execution of the program, so the possibility of them being thrown is not known when the code is written. Let's consider the following code sample.

```
package com.apress.bgn.ch4.ex;

import com.apress.bgn.ch4.hierarchy.Performer;

public class ExceptionsDemo {

    public static void  main(String... args) {
        Performer p = PerformerGenerator.get("John");

        System.out.println("TTL: " + p.getTimeToLive());
    }
}
```

Let's suppose we do not have access to the code of the `PerformerGenerator` class. We know that if we call the `get(..)` method with a name, it returns a `Performer` instance. So, we write the preceding code and try to print the performer time to live. What happens if the performer is not initialized with a proper object, because the `get("John")` method call returns null? The outcome is depicted in the next code snippet.

```
Exception in thread "main" java.lang.NullPointerException
        at chapter.four/com.apress.bgn.ch4.ex.ExceptionsDemo.
        mainExceptionsDemo.java:10
```

But if we are smart developers, or a little paranoid, we can prepare for this case, catch the exception and throw an appropriate message or perform there a dummy initialization, in case the performer instance is used in some other way later in the code.

```
package com.apress.bgn.ch4.ex;

import com.apress.bgn.ch4.hierarchy.Performer;

public class ExceptionsDemo {

    public static void main(String... args) {
        Performer p = null;//PerformerGenerator.get("John");
        try {
            System.out.println("TTL: " + p.getTimeToLive());
        } catch (Exception e) {
            System.out.println("The performer was not initialised properly
                because of: " + e.getMessage() );
        }
    }
}
```

The exception that was thrown is of type `NullPointerException`, a class that extends `RuntimeException`, so a `try/catch block` is not mandatory. This type of exceptions are called **unchecked exceptions**, because the developer is not obligated to check for them. The `NullPointerException` is the exception type Java beginner developers get a lot because they do not have the "paranoia sense" developed enough to always test objects with unknown origin before using them.

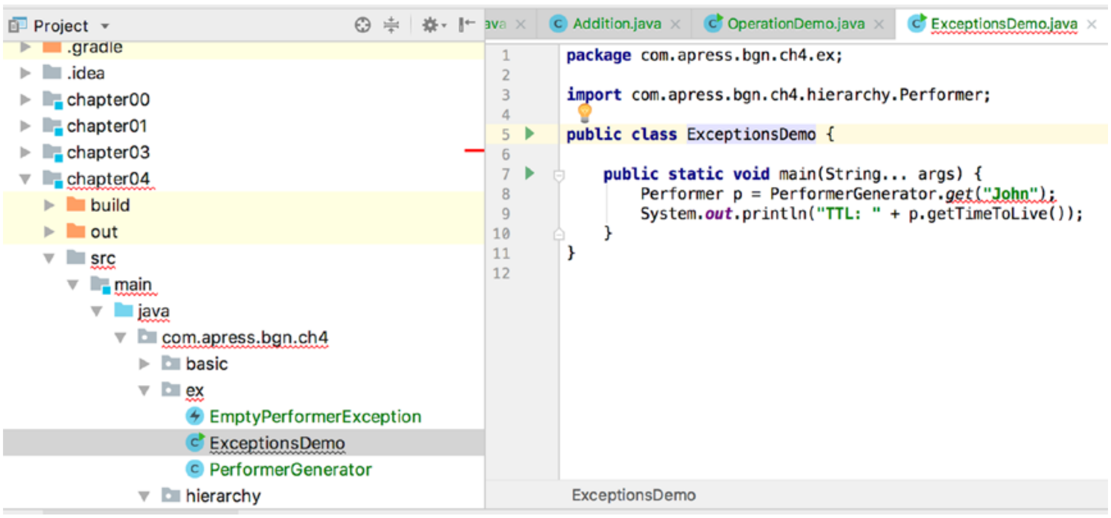
There is another type of exceptions that are called **checked exceptions**. This is any type of exception that extends `Exception`—including custom exception classes declared by the developer—that are declared as explicitly thrown by a method. In this case, when invoking that method, the compiler forces the developer to treat that exception or throws it forward. Let's use a mock implementation for `PerformerGenerator`.

```
package com.apress.bgn.ch4.ex;

import com.apress.bgn.ch4.hierarchy.Gender;
import com.apress.bgn.ch4.hierarchy.Performer;

public class PerformerGenerator {

    public static Performer get(String name)
          throws EmptyPerformerException {
        return new Performer(name,40, 1.91f, Gender.MALE);
    }
}
```

The `EmptyPerformerException` is a simple custom exception class that extends the `java.lang.exception` class.

```
package com.apress.bgn.ch4.ex;

public class EmptyPerformerException extends Exception {
    public EmptyPerformerException(String message) {
        super(message);
    }
}
```

We declared that the `get(..)` method might throw `EmptyPerformerException`; and without a `try/catch` block wrapping that method call a compiler error is thrown, as depicted in Figure 4-12.



```
package com.apress.bgn.ch4.ex;

import com.apress.bgn.ch4.hierarchy.Performer;

public class ExceptionsDemo {

    public static void main(String... args) {
        Performer p = PerformerGenerator.get("John");
        System.out.println("TTL: " + p.getTimeToLive());
    }
}
```

Figure 4-12. *Java compiler error caused by checked exception*

How do we fix it? Well, we write the code to catch it and print a relevant message.

```
package com.apress.bgn.ch4.ex;

import com.apress.bgn.ch4.hierarchy.Performer;

public class ExceptionsDemo {
    public static void main(String... args) {
        try {
            Performer p = PerformerGenerator.get("John");
            System.out.println("TTL: " + p.getTimeToLive());
        } catch (EmptyPerformerException e) {
            System.out.println("Cannot use an empty performer
               because of " + e.getMessage());
        }
    }
}
```

And since we are talking about exceptions, the `try/catch` block can be completed with a `finally` block. The contents of the `finally` block are executed if the exception is thrown further, or if the method returns normally. The only situation in which the `finally` block is not executed is when the program ends in an error.

```
package com.apress.bgn.ch4.ex;

import com.apress.bgn.ch4.hierarchy.Performer;

public class ExceptionsDemo {

    public static void main(String... args) {
        try {
            Performer p = PerformerGenerator.get("John");
            System.out.println("TTL: " +  p.getTimeToLive());
        } catch (EmptyPerformerException e) {
            System.out.println("Cannot use an empty performer!");
        } finally {
            System.out.println("All went as expected!");
        }
    }
}
```

During this book, we write code that ends in exceptional situations, so we'll have the opportunity to expand the subject when your knowledge is a little more advanced.

Generics

Until now we talked only of object types and java templates used for creating objects. But what if we would need to design a class with functionality that applies to multiple types of objects? Since every class in Java extends the `Object` class, we can create a class with a method that receives a parameter of type `Object`, and in the method we can test the object type. Take this for granted; it is covered later.

In Java 5, the possibility to use a type as parameter when creating an object was introduced. The classes that are developed to process other classes are called **generics**.

When writing Java applications, you most likely need at some point to pair up values of different types. The simplest version of a `Pair` class that can hold a pair of instances of any type is listed in the following code snippet.

```
package com.apress.bgn.ch4.gen;
public class Pair<X, Y> {

        protected X  x;
        protected Y  y;
        private Pair(X x, Y y) {
                this.x = x;
                this.y = y;
        }

        public X x() {
                return x;
        }

        public Y y() {
                return y;
        }

        public void x(X x) {
                this.x = x;
        }

        public void y(Y y) {
                this.y = y;
        }

        ...

        public static <X, Y> Pair<X, Y> of(X x, Y y) {
                return new Pair<>(x, y);
        }

    @Override public String toString() {
         return "Pair{" + x.toString() +", " + y.toString() + '}';
    }
}
```

Let's test it! Let's create a pair of `Performer` instances, a pair of a `String` and a `Performer` instance, and a pair of `Strings` to check if this is possible. The `toString()}` method is inherited from the `Object` class and overridden in the `Pair` class to print the values of the fields.

```
package com.apress.bgn.ch4.gen;

import com.apress.bgn.ch4.hierarchy.Gender;
import com.apress.bgn.ch4.hierarchy.Performer;
```

```
public class GenericsDemo {
    public static void main(String... args) {
        Performer john = new Performer("John", 40, 1.91f, Gender.MALE);
        Performer jane = new Performer("Jane", 34, 1.591f, Gender.FEMALE);

        Pair<Performer, Performer> performerPair = Pair.of(john, jane);
        System.out.println(performerPair);

        Pair<String, String> stringPair = Pair.of("John", "Jane");
        System.out.println(stringPair);

        Pair<String, Performer> spPair = Pair.of("John", john);
        System.out.println(spPair);

        System.out.println("all good.");
    }
}
```

If you execute the preceding class, you see something like the log depicted, as follows.

```
Pair{com.apress.bgn.ch4.hierarchy.Performer@1d057a39com.apress.bgn.ch4.
                 hierarchy.Performer@26be92ad}
Pair{JohnJane}
Pair{Johncom.apress.bgn.ch4.hierarchy.Performer@1d057a39}
all good.
```

The `println` method expects its argument to be a `String` instance, the `toString()` method is called on the object given if argument if the type is not `String`. If the `toString` method was not overridden, the one from the `Object` class is called that returns the fully qualified name of the object type and something called a **hashcode**, which is a numerical representation of the object.

Java Reserved Words

Table 4-2 and Table 4-3 list Java keywords that can be used only for their fixed and predefined purposes in the language. This means they cannot be used as identifiers; you cannot use them as names for variables, classes, interfaces, enums, or methods.

Table 4-2. *Java Keywords (part 1)*

Method	Description
`abstract`	Declares a class or method as abstract—as in any extending or implementing class, must provide a concrete implementation.
`assert`	Test an assumption about your code. Introduced in Java 1.4, it is ignored by the JVM, unless the program is run with "`-ea`" option.
`boolean` `byte` `char` `short` `int` `long` `float` `double`	Primitive type names
`break`	Statement used inside loops to terminate them immediately.
`continue`	Statement used inside loops to jump to the next iteration immediately.
`switch`	Statement name to test equality against a set of values known as cases.
`case`	Statement to define case values in a `switch` statement.
`default`	Declares a default case within a `switch` statement. Also used to declare default values in interfaces. And starting with Java 8, it can be used to declare default methods in interfaces, methods that have a default implementation.
`try` `catch` `finally` `throw` `throws`	Keywords used in exception handling.
`class` `interface`	Keywords used in classes and interfaces declarations.
`extends` `implements`	Keywords used in extending classes and implementing interfaces.

(*continued*)

Table 4-2. *(continued)*

Method	Description
`enum`	Keyword introduced in Java 5.0 to declare a special type of class that defines a fixed set of instances.
`const`	Not used in Java; a keyword borrowed from C where it declares constants, variables that are assigned a value, which cannot be changed during the execution of the program.
`final`	The equivalent of the `const` keyword in Java. Anything defined with this modifier, cannot change after a final initialization. A final class cannot be extended. A final method cannot be overridden. A final variable has the same value that was initialized with throughout the execution of the program. Any code written to modify final items, lead to a compiler error.

Table 4-3. *Java Keywords (part 2)*

Method	Description
`do`	Keywords to create loops:
`while`	`do{..} while(condition),`
`for`	`while(condition){..},` `for(initialisation;condition;incrementation){..}`
`goto`	Another keyword borrowed from C, but that is currently not used in Java, because it can be replaced by labeled `break` and `continue` statements
`if`	Creates conditional statements:
`else`	`if(condition) {..}` `else {..}` `else if (condition ) {..}`
`import`	Makes classes and interfaces available in the current source code.
`instanceof`	Tests instance types in conditional expressions.
`native`	This modifier indicates that a method is implemented in native code using JNI (Java Native Interface).

(continued)

Table 4-3. (*continued*)

Method	Description
`new`	Creates java instances.
`package`	Declares the package the class/interface/enum/annotation is part of and it should be the first Java statement line.
`public` `private` `protected`	Access-level modifiers for Java items (templates, fields, or methods).
`return`	Keyword used within a method to return to the code that invoked it. The method can also return a value to the calling code.
`static`	This modifier can be applied to variables, methods, blocks, and nested classes. It declares an item that is shared between all instances of the class where declared.
`stricfp`	Used to restrict floating-point calculations to ensure portability. Added in Java 1.2.
`super`	Keyword used inside a class to access members of the super class.
`this`	Keyword used to access members of the current object.
`synchronized`	Ensures that only one thread executes a block of code at any given time. This avoids a problem cause "race-condition"[3].
`transient`	Marks data that should not be serialized.
`volatile`	Ensures that changes done to a variable value are accessible to all threads accessing it.
`void`	Used when declaring methods as a return type to indicate that the method does not return a value.
`_(underscore)`	Cannot be used as an identifier starting with Java 9.

[3]A detailed article describing this problem and ways to avoid it can be found here: https://devopedia.org/race/-condition/-software

Summary

The most often used elements of the Java language were introduced in this chapter, so that nothing you find in future code samples should surprise you, and you can focus on learning the language properly.

- Syntax mistakes prevent java code from being transformed into executable code. This means the code is not compiling.
- Static variables can be used directly when declaring classes if static import statement are used.
- Java identifiers must respect naming rules.
- Comments are ignored by the compiler and there are three types of comments in Java.
- Classes, interfaces, and enums are Java components used to create objects.
- Abstract classes cannot be instantiated, even if they can have constructors.
- Interfaces could only contain method templates until Java version 8, when default methods were introduced. And starting with Java 9 they can contain full implemented methods as long as they are declared private and are being called only from default methods.
- Enums are special types of classes that can only be instantiated a fixed number of times.
- In Java, there is no multiple inheritance using classes.
- Interfaces can extend other interfaces.
- Java defines a fixed number of keywords, called **reserved keywords**, which can be used only for a specific purposes. They are covered in the previous section.