CHAPTER 5
Defining a Class in Java

In Chapter 3 we have seen that a class is a reference data type. Just like we have primitive data types in Java, which are predefined by the Java language, classes are also data types, but these are defined by a programmer. Just like we can use the primitive data in different applications, the classes defined by a programmer can also be reused in a number of applications.

The methods in a class are like the operators for the data type (class). In Exercise 2.1, we defined the class `HelloWorld` as a place holder for the Java application. A Java application is defined in a method called `main()`, so let us call those classes that contain the application (method main) as an application class. All classes need not contain `main()`, e.g. `String` class that has been used in the `HelloWorld` does not contain `main()`. It is used as a data type. So let us define classes to be used as a data type. Let us start by defining a class for representing a `Rectangle`. This could be defined as given in Listing 5.1.

Listing 5.1. Defining a class

```
1 class Rectangle {
2     // members of the class
3 }
```

5.1 VARIOUS MEMBERS WITHIN A CLASS

Now we have a new data type called `Rectangle`. An application may declare variables of this new data type as given in Listing 5.2.

Listing 5.2. Declaring reference variables of a class

```
1 class TestRectangle {
2     public static void main(String[] args) {
3         Rectangle r1, r2;
4     }
5 }
```

An instance of a class can be created in an application by using the new operator as given in Listing 5.3.

Listing 5.3. Creating an instance of a class using new

```
1 class TestRectangle {
2     public static void main(String[] args) {
3         Rectangle r1, r2;
4         r1 = new Rectangle();
5         r2 = new Rectangle();
6     }
7 }
```

5.1.1 Instance Variables

The instances of a data type may maintain some information (state). This class would require to maintain the length and the width. Let us have these values maintained as int types.

The Rectangle class could now be defined as in Listing 5.4.

Listing 5.4. Instance variables in a class

```
1 class Rectangle {
2     int length;    // instance variable
3     int width;
4 }
```

The length and width in the above class definition are known as instance variables, since every instance of the Rectangle class will have its own separate copy of length and width. Whenever an instance of Rectangle is created, the instance has space for length and width. The initial default values for these instance variables will be 0. The default values for instance variables will be 0 for all numeric types (byte, short, int, long, float, double and char), false for boolean types and null for any reference type. The instance variables of a class may be accessed in an application as in Listing 5.5.

Listing 5.5. Accessing instance variables

```
1  class TestRectangle {
2      public static void main(String[] args) {
3          Rectangle r1, r2;
4          r1 = new Rectangle();
5          r2 = new Rectangle();
6
7          // assign values to the instance variables length and width,
8          // for the instance referred to by r1.
9          r1.length = 7;
10         r1.width  = 5;
11
12         System.out.println("Length of r1 is:"+r1.length);
13     }
14 }
```

5.1.2 Methods

Let us now define an interaction with the instance of the `Rectangle` class using a method. Any method defined in a class has a return type followed by the method name. This is then followed by a list of parameter declarations in parentheses. In case the methods do not return a value, the return type must be specified as `void`. The parameter declaration is used to pass parameters to the method. It is used to accept any additional information, which may be required for the interaction. Let us define a method for interacting with the instance of `Rectangle` in order to know its area. This may be done as given in Listing 5.6.

Listing 5.6. Defining methods in a class

```
1 class Rectangle {
2     int length;   // instance variable, separate copy for every instance
3     int width;
4     int area() {  // method which can be invoked on any instance of
          this class
5         return   length *  width;
6     }
7 }
```

The `area()` method in this case does not require any information and, therefore, the parameter list is empty. The information about area is represented as an `int` value, so the return type is declared as `int`. Method implementation must have a `return` statement followed by an expression that matches the return type of the method, except for methods where the return type is `void`, in which case the method should not use an expression after the return statement. Such methods may not even contain a `return` statement. In the `area()` method the `return` statement is followed by an `int` expression. An application may invoke the method `area()` defined in `Rectangle` class as in Listing 5.7.

Listing 5.7. Invoking method on an instance

```
1  class TestRectangle {
2      public static void main(String[] args) {
3          Rectangle r1, r2;
4          r1 = new Rectangle();
5          r2 = new Rectangle();
6          r1.length = 7;
7          r1.width  = 5;
8          System.out.println("area of r1 is:"+r1.area());
9      }
10 }
```

Let us add another interaction for setting the dimensions on the instance of `Rectangle`. We can define a method called `setDimensions()` for this purpose. This interaction would require that we pass the information to the method regarding the new values of `length` and `width` that are desired, so we need parameters for the new values of `length` and `width`. This method would not return a value, so we need to declare the return type as `void`. Therefore, we define the method `setDimensions()` in the `Rectangle` class as in Listing 5.8.

Listing 5.8. Method with parameters

```
1  class Rectangle {
2       int length;    // instance variable
3       int width;
4       int area() {  // method which can be invoked on any instance of
            this class
5           return   length *  width;
6       }
7       void setDimensions(int l, int w) { // method for setting dimensions.
8           length = l;
9           width  = w;
10      }
11 }
```

This method may be used from an application with two integers as in Listing 5.9.

Listing 5.9. Invoking methods on the `Rectangle` instance

```
1 public static void main(String[] args) {
2     Rectangle r1, r2;
3     r1 = new Rectangle();
4     r2 = new Rectangle();
5     r1.setDimensions(7, 5); // setting Dimensions on r1 instance.
6     System.out.println("area of r1 is:"+r1.area()); // invoking area
          method on r1
7 }
```

Overloading methods: Now let's say we would like to set the dimensions of `Rectangle` to be such that both `length` and `width` are the same, e.g. r2.setDimensions(7, 7). Here, it may be more convenient to specify a single parameter, something like r2.setDimensions(7). We may define yet another method in the `Rectangle` class as in Listing 5.10.

Listing 5.10. `Rectangle` class with method overloading

```
1  class Rectangle {
2       int length;    // instance variable
3       int width;
4       int area() {  // method which can be invoked on any instance of
            this class
5           return   length *  width;
6       }
7       void setDimensions(int l, int w) { // method for setting dimensions.
8           length = l;
9           width  = w;
10      }
11      void setDimensions(int l) { // method for setting dimensions.
12          setDimensions(l, l);
13      }
14 }
```

The newly added method has the same name as an existing method but has a different sets of parameters. This is known as method overloading. This would now allow an application to use any of the two forms of `setDimensions()`. Depending on how many and what type of parameters are used by the invoker, the appropriate method will be used from the `Rectangle` class. So, now our class `Rectangle` has two instance variables. It has methods called `area()` and `setDimensions()`, and the `setDimensions()` method is overloaded.

Now what is the output of executing the application in Listing 5.11?

Listing 5.11. Using overloaded methods

```
1  class TestRectangle {
2      public static void main(String[] args) {
3          Rectangle r1, r2;
4          r1 = new Rectangle();
5          r2 = new Rectangle();
6          r1.setDimensions(7, 5); // setting Dimensions on r1 instance.
7          System.out.println("area of r1 is:"+r1.area()); // invoking
               area method on r1
8          r1.setDimensions(7); // using the overloaded method
9          System.out.println("area of r1 is:"+r1.area()); // invoking
               area method on r1
10         System.out.println("area of r2 is:"+r2.area()); // invoking
               area method on r2
11     }
12 }
```

It prints

```
area of r1 is:35
area of r1 is:49
area of r2 is:0
```

Note that the area of `r2` is printed as `0`. This is because by default the values of `length` and `width`, which are instance variables, are initialized to 0. Now, we may not like anybody being able to use an instance of `Rectangle` for which dimensions have not be specified. We may like to put a constraint on the creation of the instance of the `Rectangle` class, such that if anyone wants to have an instance of the `Rectangle`, then the values of `length` and `width` must be provided; otherwise `Rectangle` would not be created. This can be done by defining a constructor in the `Rectangle` class.

5.1.3 Constructors

We define a constructor by using the class name. It is written almost similar to a method, but it should not have a return type and its name is the same as the class name. So, for our `Rectangle` class we may define a constructor as in Listing 5.12.

Listing 5.12. Constructor for `Rectangle` class

```
1 Rectangle(int l, int w) {
2     setDimensions(l, w);
3 }
```

In the code given in Listing 5.12, if we mention a return type like void then it would not give a compilation error, but it would then be considered as a method, which can be invoked on any instance, and cannot be used with the new operator.

After having defined the constructor in the Rectangle class, if we now try to compile the application, which uses **new** Rectangle() to create an instance of Rectangle, it would result in a compilation error. Now the new operator could be used to invoke the constructor, which we have defined as given below:

```
1      Rectangle r1, r2;
2      r1 = new Rectangle(7, 5);  // invoking the constructor to
           create a new instance.
```

Earlier the expression **new** Rectangle() was working fine, but now after the introduction of the constructor as in Listing 5.15, it would not compile. In Java, every class has at least one constructor. When we do not define a constructor for a class, then the compiler introduces a public constructor with no arguments. In Listing 5.10, when we did not define a constructor for the Rectangle class, the compiler had defined a public constructor with no argument. However, if we define a constructor in our class definition, then this no argument constructor would not be defined by the compiler since it finds a constructor in the class definition.

Constructors are mainly used to initialize the instance variables. They are always used with the new operator to allocate and initialize a new instance of the class.

EXERCISE 5.1 Define a class called Account to represent a bank account. Every Account has an account number, the name of the account holder and a balance. Define an appropriate constructor. Define the methods for retrieving the values of the account number, name and the balance for a given Account instance. Also define methods called deposit and withdraw, which require the amount as parameter and a method called display, which displays the states of various instance variables of the instance on standard output. The withdraw method may return a boolean, depending on whether the withdrawal is done successfully or not. The withdrawal will not succeed in case the Account does not have sufficient balance. Also define a separate class called TestAccount, with a main method, to test the methods and constructors of the Account class.

The Account class may be defined as given in Listing 5.13.

Listing 5.13. Account.java

```
1
2  class Account {
3
4      int accountNumber;
5      String name;
6      double balance;
7
8      Account(int acno, String n, double openBal) {
9          this.accountNumber = acno;
10         this.name = n;
```

```
11          this.balance = openBal;
12      }
13
14      int getAccountNumber() {
15          return this.accountNumber;
16      }
17
18      String getName() {
19          return this.name;
20      }
21
22      double getBalance() {
23          return this.balance;
24      }
25
26      void deposit(double amt) {
27          this.balance += amt;
28      }
29
30      boolean withdraw(double amt) {
31          if (this.balance < amt) {
32              return false;
33          }
34          this.balance -= amt;
35          return true;
36      }
37
38      void display() {
39          System.out.println("Account:"+this.accountNumber+","+this.
                name+","+this.balance);
40      }
41  }
```

Overloading constructors: Just like we can overload methods, even the constructors may also be overloaded. In the case of the `Rectangle` class above, we may like to allow the creation of a `Rectangle` instance even by specifying a single `int` value, in which case we may like to invoke the earlier constructor with two parameters. This can be done as given in Listing 5.14.

Listing 5.14. Using this to invoke an overloaded constructor

```
1 Rectangle(int l) {  // overloading of constructor.
2     this(l, l);     // invoking the constructor with 2 parameters.
3 }
```

Invoking a constructor of the same class using this: In the above constructor, we have used this(l, l). Here `this` is not a method, it is a keyword in Java. `this` is used in a constructor to invoke the constructor of the same class in case of constructor overloading. In a constructor we can invoke another constructor of the same class by using the keyword `this` followed by

the list of parameter values, similar to invoking a method. The keyword `this` followed by a list of parameters can be used to invoke another constructor of the same class, only from within a constructor, and whenever it is used, it has to be the first statement in the constructor. So now the class `Rectangle` may look as in Listing 5.15.

Listing 5.15. `Rectangle` class with constructors

```
1  class Rectangle {
2      int length;    // instance variable
3      int width;
4      int area() {  // method which can be invoked on any instance of
           this class
5          return   length *  width;
6      }
7      void setDimensions(int l, int w) { // method for setting dimensions.
8          length = l;
9          width  = w;
10     }
11     void setDimensions(int l) { // method for setting dimensions.
12         setDimensions(l, l);
13     }
14     Rectangle(int l, int w) { // constructor with 2 parameters
15         setDimensions(l, w);
16     }
17     Rectangle(int l) {  // overloading of constructor.
18         this(l, l);     // invoking the constructor with 2 parameters.
19     }
20 }
```

Let us now use this `Rectangle` class in an application as given in Listing 5.16.

Listing 5.16. Using overloaded constructors

```
1  class TestRectangle {
2      public static void main(String[] args) {
3          Rectangle r1, r2;
4          r1 = new Rectangle(7, 5);
5          r2 = new Rectangle(7);
6          System.out.println("area of r1 is:"+r1.area()); // invoking
               area method on r1
7          System.out.println("area of r2 is:"+r2.area()); // invoking
               area method on r2
8          r2 = r1;
9          r1.setDimensions(9, 5);
10         System.out.println("area of r2 is:"+r2.area()); // invoking
               area method on r2
11     }
12 }
```

In Listing 5.16 in Line 3, two references r1 and r2 are declared. Line 4 uses the constructor with two parameters to create an instance of `Rectangle` and it lets `r1` refer to the instance. Line 5 uses the constructor with one parameter and creates an instance and assigns it to `r2`. Lines 6 and 7 print the area of instances referred by `r1` and `r2`, respectively. Now, Line 8 assigns `r1` to `r2`, which would result in `r2` stopping to refer to the instance created in Line 5, and start referring to the instance created in Line 4, the one currently being referred by `r1`. Line 9 sets the dimensions of the `Rectangle` referred by `r1` to length=9 and width=5. Now when we print the area of `r2` in Line 10, it prints the area according to the dimensions set in Line 9 for `r1`. This is because in Line 8, we made `r1` and `r2` refer to the same instance.

5.1.4 The `finalize` Method

In Line 8, when we assigned `r1` to `r2`, what happened to the instance that `r2` was initially referring, the one that was created in Line 5? After execution of Line 8, that instance which was initially referred by `r2` is now eligible for garbage collection, and may get garbage collected any time later, as decided by the garbage collector. In Java only the garbage collector can deallocate any instance. An instance would be eligible for garbage collection only when there are no references referring to it from the application. The garbage collector works parallel to the application, and its only job is to look for instances that cannot be used from the application (the ones that do not have any references from application threads). Now, when the garbage collector decides to deallocate any instance, it looks for a special method in that class definition for the instance called `finalize()`, and would invoke it just before deallocation. Therefore, in a class definition we may define a method called `finalize()`, which is not meant to be invoked from the application, but is meant for use by the garbage collector, before the garbage collector deallocates any instance of that class. The method signature for the `finalize` is as follows:

```
1     protected void finalize()
```

Therefore, in the life of any instance the constructor is like the first method that is invoked on it, and cannot be invoked any other time, and `finalize()` is the last method invoked in the life of any instance of a class. Now, coming back to the `Rectangle` class definition, suppose we would like to have a mechanism to maintain the count of instances of `Rectangle` in the JVM, i.e. we would like to know at any point of time as to how many objects of the `Rectangle` have been created and not yet been deallocated by the garbage collector. This can probably be achieved by having some variable, let us say `count` of type `int`, which can be incremented from the constructor and have a method `finalize()` defined in the `Rectangle` class to decrement the `count` as given in Listing 5.17.

Listing 5.17. Using count in constructor and `finalize` method

```
1 Rectangle(int l, int w) {
2     count++;     // increment the value of count from constructor.
3     setDimensions(l, w);
4 }
5 ...
6 protected void finalize() {
```

```
7       count--;    // decrement the value of count, to be used by
            garbage collector.
8   }
```

In the above class, where is count declared? Do we declare it as an instance variable along with `length` and `width`? No, we can't have `count` as an instance variable, since it would mean that we have a separate copy of `count` maintained by each instance. We want exactly one copy of `count`, whatever be the number of instances of `Rectangle`.

5.1.5 `static` Variables and `static` Methods

We can define `count` as a class variable, i.e. a variable with one copy per class definition and not dependent on any instance of the class. We define a class variable in a class definition by declaring a member variable as `static`. Therefore, we now introduce another member in the `Rectangle` class called `count` as follows:

```
1       static int count;
```

With the above variable declaration in the `Rectangle` class and the previous change in the constructor and the `finalize()` method, we would be able to manage the count of `Rectangle` instances in memory in the `static` variable called `count`. Note that we increment the `count` only in the constructor with two parameters and not from the constructor with one parameter since the other constructor calls the constructor with two parameters by using `this`. Now, from an application, we would be able to access the static variable also known as class variable by using the class name, as shown below:

```
Rectangle.count
```

Any `static` members of a class are accessed by using the class name, as is shown in the above example for the `static` variable `count`. The `static` variables are allocated as soon the class is loaded. Every class is always loaded first, before it can be used in an application. Therefore, the `count` variable would be allocated even before any instance of the `Rectangle` class is created. Similar to the instance variables the default initial value for the class variables is also 0 for numeric types, false for `boolean` and null for the reference types.

Just like we have a `static` variable as a member of a class definition, we can also have a `static` method in a class definition. These `static` methods, like the static variables, are not dependent on an instance of the class. For example, in the `Rectangle` class above we may provide a method to return the value of the `static` variable count as in Listing 5.18.

Listing 5.18. Static method `getCount`

```
1 static int getCount() {
2     return count;
3 }
```

Similar to accessing the static variable `count`, even this method may be accessed from an application by using the class name as follows:

```
Rectangle.getCount()
```

Now, coming to the increment of `count`, which is to be done whenever any instance of `Rectangle` is created. In the case of the `Rectangle` class above, this increment is currently being done from the constructor with two parameters. This increment is to be carried out no matter which constructor is used.

EXERCISE 5.2 Modify the `Account` class in Exercise 5.1 such that the `accountNumbers` are maintained automatically, i.e. the `Account` class should also have a constructor where the account number is not a parameter and is maintained automatically. Also update the `TestAccount` class to test the new constructor with two parameters.

The `Account` class of Listing 5.13 may be modified as given in Listing 5.19.

Listing 5.19. Account with two constructors

```
1
2 class Account {
3
4     int accountNumber;
5     String name;
6     double balance;
7
8     Account(int acno, String n, double openBal) {
9         this.accountNumber = acno;
10        this.name = n;
11        this.balance = openBal;
12    }
13
14    int getAccountNumber() {
15        return this.accountNumber;
16    }
17
18    String getName() {
19        return this.name;
20    }
21
22    double getBalance() {
23        return this.balance;
24    }
25
26    void deposit(double amt) {
27        this.balance += amt;
28    }
29
30    boolean withdraw(double amt) {
31        if (this.balance < amt) {
```

```
32              return false;
33          }
34          this.balance -= amt;
35          return true;
36      }
37
38      void display() {
39          System.out.println("Account:"+this.accountNumber+","+this.
                name+","+this.balance);
40      }
41
42      static int lastAccountNumber = 1000;
43
44      Account(String n, double openBal) {
45          this(++lastAccountNumber, n, openBal);
46      }
47  }
```

5.1.6 Initializer Block

In a class definition, we can have a member block with no name. Such a block is known as the initializer block. An initializer block is never invoked by any application directly since it does not have any name. However, it is always invoked on an instance as soon as it is allocated. It is invoked just before the constructor is invoked on the instance. In the case of the `Rectangle` class above, we can remove the increment of count from the constructor and define an initializer block for the class as given in Listing 5.20.

Listing 5.20. Using the initializer block

```
1 Rectangle(int l, int w) {  // remove the increment of count from
      the constructor
2     setDimensions(l, w);
3 }
4 ...
5 {       // initializer block, executed whenever any instance is created.
6     count++;
7 }
```

The initializer block in a class is invoked whenever any of the constructor for the class is invoked. This block is always executed before the constructor code is executed. The initialization process for an instance is thus executed in the following sequence. Whenever any constructor is invoked with the new operator to create a new instance, first the space is allocated for the instance depending on the instance variables declared in the class. When this space is allocated, its instance variables have the default initial value according to the type of variable, i.e. numeric types will be 0, `boolean` types will be `false` and reference types will be `null`; now in case the instance variable declaration has an assignment, then such an assignment will be executed on the instance; for example, in case of the `Rectangle` class, if the instance variable `length` were declared as

```
1     int length = 7;      // instance declaration with assignment
```

then the assignment length = 7 would be executed initially. Now after executing instance variable initialization, the initializer block would be executed, and then it is followed by execution of the specific constructor code that is invoked.

Suppose, for the time being we changed the code in the `Rectangle` class relating to the instance declaration, the constructor with two parameters and the initializer block are as given in Listing 5.21.

Listing 5.21. Code to track the flow during instance creation

```
1  int length = 7;  // instance declaration with assignment.
2  ...
3  Rectangle(int l, int w) {
4      System.out.println("length at the start of constructor:"+length);
5      count++;
6      setDimensions(length+1, w); // increase the current value of
           length by 1.
7      System.out.println("length at the end of constructor:"+length);
8  }
9  ...
10 {
11     System.out.println("length at the start of initializer block:"+
           length);
12     length += 5;
13     System.out.println("length at the end of initializer block:"+length);
14 }
```

We then use the constructor with two parameters from an application as given in Listing 5.22.

Listing 5.22. Invoking a constructor to track the flow

```
1 public static void main(String[] args) {
2     Rectangle r1 = new Rectangle(7, 5);
3 }
```

Then the output generated is as follows:

```
length at the start of initializer block:7
length at the end of initializer block:12
length at the start of constructor:12
length at the end of constructor:19
```

The above output shows the sequence in which the initialization code is executed whenever any constructor is used with the new operator from an application.

5.1.7 Class Initializer Block

Just like we have the constructor for initializing the instance variables, we use the class initializer block to initialize the class variables. A class initializer block is created just like the initializer block, but it is declared to be `static`. We also call this block as the `static` block, e.g. in the `Rectangle` class, we may declare a `static` block as given in Listing 5.23.

Listing 5.23. Using a static block to initialzie static variables

```
1 static {
2     count = 0;
3     System.out.println("Inside a class initializer block");
4 }
```

Just like the initializer block, the class initializer block does not have a name and is not invoked directly by an application. This block is automatically executed whenever the class is loaded. A class is normally loaded only once at the runtime. The class would be loaded by the JVM only if the class is used in an application. A mere declaration of variables would not result in the class being loaded. Therefore, if we have the code in an application as given in Listing 5.24, then the loading of class would be done in Line 3 the first time we use any of the members of the class. A mere declaration in Line 2 does not result in the loading of the class. In Line 3 when the class is loaded, it would first execute the static block. Now let's revert the changes and have the `Rectangle` class as given in Listing 5.25.

Listing 5.24. Class loading

```
1 public static void main(String[] args) {
2     Rectangle r1, r2; // does not result in loading of Rectangle class.
3     int rectangleCount = Rectangle.getCount();  // results in loading
          of the Rectangle.
4                                 // ie. execution of static block will precede
5                                 // execution of the getCount method.
6     r1 = new Rectangle(7); // results in the execution of initializer
          block first
7                                 // followed by execution of the constructor.
8 }
```

Listing 5.25. `Rectangle` class with various members

```
1  class Rectangle {
2      static int count;    // class variable
3      int length;          // instance variable
4      int width;
5      int area() {         // method definition
6          return length * width;
7      }
8      void setDimensions(int l, int w) {
9          length = l, width = w;
10     }
```

```
11      void setDimensions(int l) { // method overloading
12          setDimensions(l, l);
13      }
14      Rectangle(int l, int w) {    // constructor
15          setDimensions(l, w);
16      }
17      Rectangle(int l) {  // constructor overloading
18          this(l, l);
19      }
20      protected void finalize() { // method executed by garbage collector
            before deallocation
21          count --;
22      }
23      static int getCount() {      // static method
24          return count;
25      }
26      {               // initializer block;
27          count++;
28      }
29      static {        // class initializer block
30          count = 0;
31      }
32  }
```

The above listing shows the different kinds of members that can be declared in a class definition. Over and above these members, we can also have nested class and interfaces, which are also members of a class, the nested classes and interfaces will be covered in Chapter 12.

Now, in the above class definition, what would be the change in the body of the `setDimensions()` method in case we use `a` and `b` as parameter variables instead of `l` and `w`. We would simply replace `l` with `a` and `w` with `b` within the body of the method as given in Listing 5.26.

Listing 5.26. `setDimensions` method

```
1 void setDimensions(int a, int b) {
2     length = a;
3     width  = b;
4 }
```

Accessing current instance using `this`: Now instead of using `a` and `b`, if we use `length` and `width` as the parameter variables, then what would be the change? Do we replace the code like in Listing 5.27?

Listing 5.27. Parameter variable name clash with instance variable

```
1 void setDimensions(int length, int width) {
2     length = length;
3     width = width;
4 }
```

This would not give a compilation error, but it does not work. The instance variables length and width are not initialized and remain 0 only. Now let us examine the method again. In the above method, length is the parameter variable. The two statements in the method are meaningless since they merely assign the value to self, i.e. the parameter variable. In the earlier case when we used l and w as the parameter variables and try to look at the method definition, we find that we do not have any variables called length and width as a parameter or a local variable in the method. However, we were allowed to use them.

This happened because the compiler made a modification on our behalf. When the compiler came across the length in the method definition, it did not find the variable being available in the method. Now instead of reporting it as an error the compiler then went about looking if we have any variable length as a member of the class. It found that length is an instance variable and so it replaced the variable length with **this**.length. What is this? this is a variable available throughout any class definition, except for the static part of the class. The difference between the static and non-static methods and blocks is that static context does not have this. So, this is the difference between static and non-static. this is a reference to the current instance. Therefore, when we used the instance variables directly in the methods or constructors the compiler replaced it with **this**.length and **this**.width in case of the Rectangle class having length and width as the instance variables. So in case of the setDimensions() method having a parameter name, which is the same as an instance variable, the compiler would treat the usage of length and width within the method as the parameter variable. The code in Listing 5.28 shows how to use an instance variable whose name clashes with a parameter or a local variable.

Listing 5.28. Accessing the instance variable with this

```
1 void setDimensions(int length, int width) {
2     this.length = length; // accessing the instance variable using this.
3     this.width = width;
4 }
```

Therefore, this is like a local variable available to all non-static methods, constructors and the initializer block, and is the reference to the instance of the current class. So, when we define a method like setDimensions(), which is a non-static method with two parameters, there is a third implicit parameter as given in Listing 5.29.

Listing 5.29. Visualizing this as a parameter to a method

```
1 void setDimensions(final Rectangle this, int length, int width) {
2 ...  // the first parameter is an implicit parameter and should not
      be declared
3 }
```

Note: this is a keyword in Java, and one should not be declaring it as a variable as shown in the code in Listing 5.29, it is used in the code just to visualize the implicit variable called this.

In many organizations, it is also part of the coding convention to explicitly use `this` in methods and constructors whenever we are using an instance variable. This way, there would be less chances for it to have a name clash with any other local variable, and it also clarifies immediately that an instance variable is being referred. So, in case of `Rectangle` class, we may update the methods and use **`this`**.length and **`this`**.width instead of `length` and `width`.

5.2 LOOKING AT THE ENTIRE CLASS

Finally the class `Rectangle`, which we have developed during the course of this chapter, would look similar to Listing 5.30.

Listing 5.30. The entire `Rectangle` class

```
1  class Rectangle {
2      static int count;    // class variable
3      int length;          // instance variable
4      int width;
5      int area() {         // method definition
6          return this.length * this.width;
7      }
8      void setDimensions(int l, int w) {
9          this.length = l, this.width = w;
10     }
11     void setDimensions(int l) { // method overloading
12         setDimensions(l, l);
13     }
14     Rectangle(int l, int w) {   // constructor
15         setDimensions(l, w);
16     }
17     Rectangle(int l) {  // constructor overloading
18         this(l, l);
19     }
20     protected void finalize() { // method executed by garbage collector
           before deallocation
21         count --;
22     }
23     static int getCount() {     // static method
24         return count;
25     }
26     {               // initializer block;
27         count++;
28     }
29     static {        // class initializer block
30         count = 0;
31     }
32 }
```

Looking at the full picture of a class, the members that can be put in a class definition are as follows:

Instance variables are the variables whose separate copy is created in every instance of the class. In the `Rectangle` class, we declared two instance variables, `length` and `width`, because every instance of `Rectangle` will have a separate value for `length` and `width`. This variable does not exist without an instance. The instance variables decide the structure of the class.

Methods are the ways in which interaction will be allowed with the instance of this class. These methods are always invoked on a target instance of the class. These cannot be used without an instance. In a class definition, we can have more than one method with the same name but they cannot have the same type and number of parameters. That is, methods in a class can be overloaded.

Constructors are used to initialize the instance variables. These are invoked on an instance whenever an instance is created using the `new` operator. The constructor is like a special method that is invoked as the first method invocation on an instance of the class. Constructors cannot be invoked on an instance like other methods. It is only used with the `new` operator.

`static` variables also known as class variables are not a part of any instance of the class. There is only one copy of these in the JVM, which comes into existence only when the class is loaded. These variables exist without depending on any instance of the class. Even if there are no instances of a given class, the class variable can still exist.

`static` methods also known as class methods are not dependent on any instance of the class. Unlike the non-static methods, these methods do not have a target instance and is invoked by using the class name. These methods do not have the implicit current object called `this`, which is available in the non-static part of the class.

Initializer block is used to define the activity that is required to be carried out whenever any instance is created for the class. There can be any number of initializer blocks in a class definition. These would be combined and treated as one initializer block. This code is executed just before any of the constructor code is executed.

Class initializer block is used to initialize the class variables. This block is executed only once when the class is loaded, and similar to the initializer block in a class definition we can have any number of class initializer blocks, which would be combined and treated as one.

LESSONS LEARNED

- A class is a reference data type, and instances of a class can be created using the `new` operator.
- The instance variables in a class define the structure of instances of the class. There is a separate copy of instance variable for each instance of the class.
- Methods in a class are used to define the possible interactions with instances of the class.
- The constructor in a class defines the first interaction that is required to be executed on the instance of a class as soon as it is created. Constructors do not have a return type. All classes have at least one constructor. If no constructor is specified for a class, then the compiler would define a `public` no arguments constructor. A constructor is normally used to initialize instance variables.
- The methods and constructors in a class can be overloaded.

3. Explain the uses of keywords `this` and `static`.
4. What are the various elements that can be defined within a class definition? Briefly explain each of them.
5. Explain the use of the `finalize()` method.
6. Explain the overloading of methods and constructors with an example.
7. Explain the process of an instance creation whenever a constructor is invoked by an application.
8. Define a class called `Product`; each product has a name, a product code and manufacturer name. Define variables, methods and constructors for the `Product` class.