# Java in Context

# 24

**Objectives**:

*By the end of this chapter you should be able to*:

- *provide a brief history of the development of the Java language*;
- *identify the potential problems with* ***pointers***, ***multiple inheritance*** *and* ***aliases***;
- *develop* `clone` *methods and* ***copy constructors*** *to avoid the problem of aliases*;
- *identify* ***immutable objects***;
- *explain the benefits of Java's* ***garbage collector***.

## 24.1 Introduction

Originally named *Oak*, Java was developed in 1991 by Sun Microsystems. The Java technology was later acquired by Oracle™. Originally, the intention was to use Java to program consumer devices such as video recorders, mobile phones and televisions. The expectation was that these devices would soon need to communicate with each other. As it turned out, however, this concept didn't take off until later. Instead, it was the growth of the Internet through the World Wide Web that was to be the real launch pad for the language.

The Java technology was acquired by Oracle™ in 2010, but the original motivation behind its development explains many of its characteristics. In particular, the **size** and **reliability** of the language became very important.



Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_24

## 24.2 Language Size

Generally, the processor power of a system controlling a consumer device is very small compared with that of a PC; so the language used to develop such systems should be fairly compact. Consequently, the Java language is relatively small and compact when compared with other traditional languages. At the time Java was being developed, C++ was a very popular programming language. For this reason the developers of Java decided to stick to conventional C++ syntax as much as possible. Consequently Java syntax is very similar to C++ syntax.

Just because the Java language is relatively small, however, does not mean that it is not as powerful as some other languages. Instead, the Java developers were careful to remove certain language features that they felt led to common program errors. These include the ability for a programmer to create **pointers** and the ability for a programmer to develop **multiple inheritance** hierarchies.

### 24.2.1 Pointers

A pointer, in programming terms, is a variable containing an address in memory. Of course Java programmers can do something very similar to this—they can create *references*. Figure 24.1 repeats an example we showed you in Chap. 7.

In Fig. 24.1, the variable `myOblong` contains a reference (address in memory) of an `Oblong` object. The difference between a *reference* and a *pointer* is that the *programmer* does not have control over which address in memory is used—the *system* takes care of this. Of course, internally, the system creates a pointer and controls its location. In a language like C++ the programmer can directly

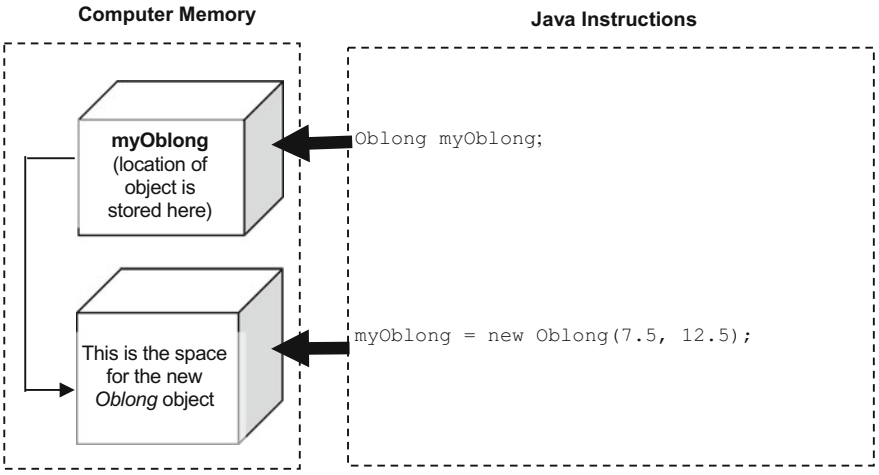


**Fig. 24.1** An object variable in Java contains a reference to the object data

manipulate this pointer (move it along and back in memory). This was seen as giving the programmer greater control. However, if this ability is abused, critical areas of memory can easily be corrupted. For this reason the Java language developers did not allow users to manipulate pointers directly.

### 24.2.2 Multiple Inheritance

Inheritance is an important feature of object-oriented languages. Many object-oriented languages, such as C++ and Eiffel, allow an extended form of inheritance known as **multiple inheritance**. When programming in Java, a class can only ever inherit from at most one superclass. Multiple inheritance allows a class to inherit from more than one superclass (see Figs. 24.2 and 24.3).

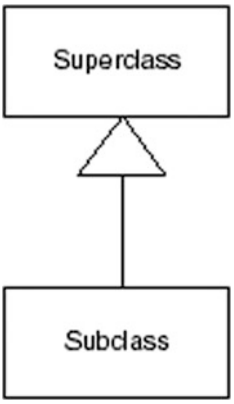


**Fig. 24.2** Single inheritance

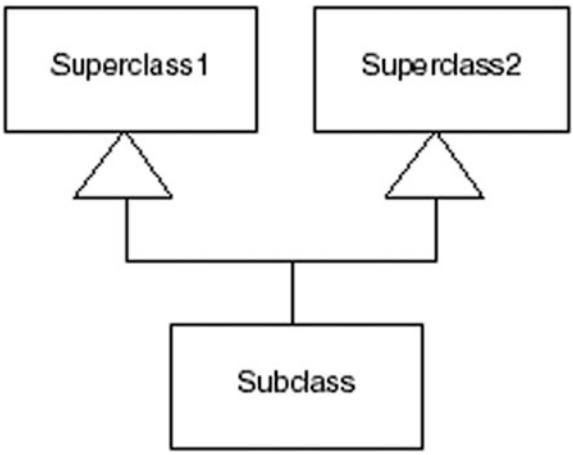


**Fig. 24.3** Multiple inheritance

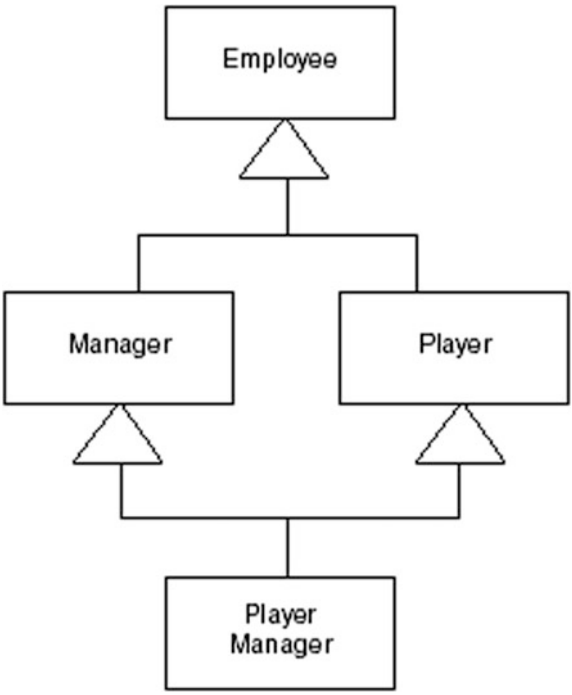


**Fig. 24.4** A combination of single and multiple inheritance

As discussed in Chap. 13, multiple inheritance can lead to a variety of problems. The Java developers decided not to allow multiple inheritance for two reasons:

- it is very rarely required;
- it can lead to very complicated inheritance trees, which in turn lead to programming errors.

As an example of multiple inheritance, consider a football club with various employees. Figure 24.4 illustrates an inheritance structure that might be arrived at.

Here, a `PlayerManager` inherits from both `Player` and `Manager`, both of which in turn inherit from `Employee`! As you can see this is starting to get a little messy. Things become even more complicated when we consider method overriding. If both `Player` and `Manager` have a method called `payBonus`, which method should be called for `PlayerManager`—or should it be overridden? This is sometimes referred to as the **diamond problem** given the diamond like shape of the problematic design (as illustrated in Fig. 24.4).

Although Java disallows multiple inheritance it does offer a type of multiple inheritance—interfaces. As we have seen in previous chapters, a class can inherit from only one base class in Java but can implement many interfaces. Up to Java 8 this meant that the diamond problem could not arise as interfaces could contain **`abstract`** methods only.

Since Java 8, however, interfaces can contain **`default`** methods. As discussed in Chap. 13, **`default`** methods are regular methods that contain an implementation and reside in interfaces. These methods are automatically inherited by classes that implement these interfaces.

You might think that this could potentially lead to the diamond problem once again if we implement two or more interfaces that contain **`default`** methods with the same name. For example, let us look at the outline of a `PlayerInterface` that contains a **`default`** `payBonus` method:

```
public interface PlayerInterface
{
    // other regular abstract methods can be included here

    // a default method has an implementation
    default double payBonus()
    {
        return 1000;
    }
}
```

You can see how we add a **`default`** method into an interface. We use the keyword **`default`** and provide an implementation. In our implementation we have given a player a bonus of 1000. It is assumed all **`default`** methods are **`public`**, so we do not need to add this scope. Now consider a `ManagerInterface` that also contains a **`default`** `payBonus` method:

```
public interface ManagerInterface
{
    // other regular abstract methods can be included here

    // this default method has the same name as the default method in the PlayerInterface
    default double payBonus()
    {
        return 2000;
    }
}
```

You can see a manager has been given a bonus of 2000. Now, consider the following `PlayerManager` class that attempts to implement both of these interfaces:

```
public class PlayerManager implements PlayerInterface, ManagerInterface
{
   // implement abstract methods of PlayerInterface and ManagerInterface
}
```

If all we include in this `PlayerManager` class are implementations for the **`abstract`** methods contained in both the given interfaces this class will **not compile**. The reason for this is to avoid the diamond problem, as it would not be clear which version of `payBonus` to inherit. To resolve this Java insists that we override the `payBonus` method with an implementation of our own. Here is one possible solution:

```
public class PlayerManager implements PlayerInterface, ManagerInterface
{
   // implement abstract methods of PlayerInterface and ManagerInterface

    @Override
     public double payBonus()
     {
         return 3000;
     }
}
```

In this case we have given a player manager a bonus of 3000. Note when overriding the `payBonus` method we must mark this as a **`public`** (unlike **`default`** methods in interfaces which are always assumed to be **`public`** but not necessarily marked as **`public`**).

Now we have overridden the `payBonus` method there is no conflict to resolve and the given class will compile. We can use either (or both) of the inherited `payBonus` implementations when overriding these methods. We do so by making using the **`super`** keyword along with the interface name. For example, we might generate a player manager bonus by adding together the player bonus and the manager bonus as follows:

```
public class PlayerManager implements PlayerInterface, ManagerInterface
{
   // implement abstract methods of PlayerInterface and ManagerInterface

    @Override
     public double payBonus()
     {
        // we can access the inherited payBonus methods when overriding these methods
        return  PlayerInterface.super.payBonus()+ ManagerInterface.super.payBonus();
     }
}
```

## 24.3 Language Reliability

The Java language developers placed a lot of emphasis on ensuring that programs developed in Java would be reliable. One way in which they did this was to provide the extensive exception handling techniques that we covered in Chap. 14. Another way reliability was improved was to remove the ability for programmers to directly manipulate pointers as we discussed earlier in this chapter. Errors arising from pointer manipulation in other languages are very common. A related problem, however, is still prevalent in Java but can be avoided to a large extent. This is the problem of **aliasing**.

### 24.3.1 Aliasing

Aliasing occurs when the *same* memory location is accessed by variables with *different* names. As an example, we could create an object, `obj1`, of the `Oblong` class as follows:

```
Oblong obj1 = new Oblong (10, 20);
```

We could then declare a new variable, `obj2`, which could reference the same object:

```
Oblong obj2 = obj1;
```

Here `obj2` is simply a different name for `obj1`—in other words an **alias**. The effect of creating an alias is illustrated in Fig. 24.5.

In practice a programmer would normally create an alias only with good reason. For example, let us assume we have an array of `BankAccount` objects called `accountList` and we wish to overwrite one `BankAccount` in the list with the adjacent `BankAccount`. We are able to make good use of aliasing by assigning an object reference to a different object with a statement like:

```
accountList[i] = accountList[i+1];
```

After this instruction, `accountList[i]` is pointing to the same object as `accountList[i+1]`. In this case that was the intention. However, a potential problem with a language that allows aliasing is that it could lead to errors arising inadvertently. Consider for example a `Customer` class that keeps track of just two bank accounts. Here is the outline of that class:

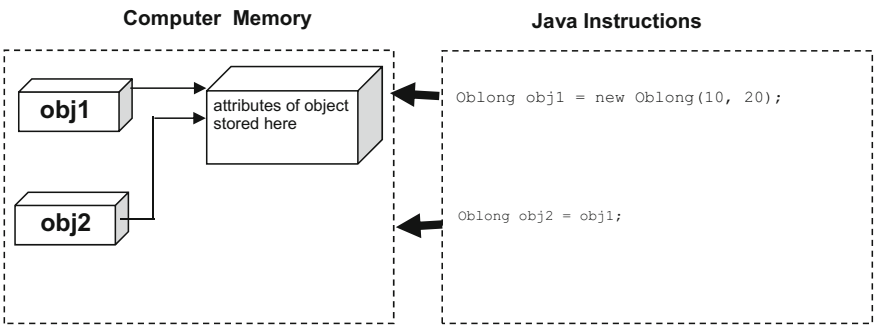


**Fig. 24.5** Copying an object reference creates an alias

```
public class Customer
{
    // two private attributes to hold bank account details
    private BankAccount account1;
    private BankAccount account2;

    // more code here

    // two access methods
    public BankAccount getFirstAccount()
    {
        return account1;
    }

    public BankAccount getSecondAccount()
    {
        return account2;
    }
}
```

Consider the methods `getFirstAccount` and `getSecondAccount`. In each case we have sent back a reference to a **`private`** attribute, which is itself an object. We did this to allow users of this class to interrogate details about the two bank accounts, with statements such as:

```
BankAccount tempAccount = someCustomer.getFirstAccount();
System.out.println("balance of first account = "+tempAccount.getBalance());
```

Let us assume that this produced the following output:

```
balance of first account = 250.0
```

This is fine, but the `tempAccount` object, that we have just created, is now an alias for the **`private`** `BankAccount` object in the `Customer` class. It can be used to manipulate this **`private`** `BankAccount` object without going through any `Customer` methods. To demonstrate this let us withdraw money from the alias:

```
tempAccount.withdraw(100); // withdraw 100 from alias
```

Now let us go back and examine the bank account in the `Customer` class:

```
double balance = someCustomer.getFirstAccount().getBalance();
System.out.println("balance of first account = " + balance);
```

In this case we have retrieved the first bank account, and its balance in one instruction. We then display this balance, giving the following output:

```
balance of first account = 150.0
```

The balance of this internal account has been reduced by 100 without the `Customer` class having any control over this! From this example you can see how dangerous aliases can be.

There are a few examples in this book where we have returned references to **`private`** objects, but we have been careful not to take advantage of this by manipulating **`private`** attributes in this way. However, the important point is that they *could* be manipulated in that way. In order to make classes extra secure (for example, in the development of critical systems), aliasing should be avoided.

The problem of aliases arises when a copy of an object's *data* is required but instead a copy of the object's *reference* is returned. These two types of copies are sometime referred to as *deep copy* (for a copy of an object's data) and *shallow copy* (for a copy of an object's reference). By sending back a shallow copy, the original object can be manipulated, whereas a deep copy would not cause any harm to the original object.

In order to provide such a deep copy, a class should define a method that returns an exact copy of the object data. Such a method exists in the `Object` class, but this should be overridden in any user-defined class. The method is called `clone`. We want to send back copies of `BankAccount` objects, so we need to include a `clone` method in the original `BankAccount` class.

### 24.3.2 Overriding the *clone* Method

You have seen examples of overriding `Object` methods before. In Chap. 15, for instance, we overrode the `toString` and `hashCode` methods in the `Object` class. There is one important difference, however, between the `clone` method and other `Object` methods such as `hashCode` and `toString`. The `clone` method is declared as `protected` in the `Object` class, whereas methods such as `hashCode` and `toString` are declared as `public`.

Methods which are **`protected`** can only be called from within the same package (`Object` is in the `java.lang` package), or *within* subclasses. Methods which are **`protected`** are *not* part of the external interface of a class.

So if we wish to provide a `clone` method for any class, we are *forced* to override the `clone` method from `Object`. When we override this method we must make it **`public`** and not **`protected`**. When overriding methods you are able to give them wider access modifiers but not less—so a **`protected`** method can be overridden to be **`public`**, but not vice versa. The return type of the `clone` method is always `Object`:

```
@Override

// clone methods you write must have this interface
public Object clone() // must be a public method
{
    // code goes here
}
```

There were sound security reasons for the Java developers forcing you to override the `clone` method if you wish objects of your classes to be cloned, rather than allow objects of *all* classes to use the `clone` method in the `Object` class; because you might be developing a class in which you did not want objects of that class to be cloned.

However, we *do* want to provide the original `BankAccount` class with a `clone` method. Such a method would allow the `Customer` class to send back clones of `BankAccount` objects, rather than aliases as it is currently doing. Here is the outline of the `BankAccount` class with one possible implementation of such a method:

```
public class BankAccount
{
    // private attributes as before
    private String accountNumber;
    private String accountName;
    private double balance;

    // previous methods go here

    // now provide a clone method
    public Object clone()
    {
        // call contsructor to create a new object identical to this object
        BankAccount copyOfThisAccount = new BankAccount (accountNumber, accountName);
        /* after this the balance of the two bank accounts might not be the same,
           so copy the balance as well */
        copyOfThisAccount.balance = balance;
        // finally, send back this copy
        return copyOfThisAccount;
    }
}
```

Notice that in order to set the balance of the copied bank account we have directly accessed the **`private`** `balance` attribute of the copy:

```
copyOfThisAccount.balance = balance;
```

This is perfectly legal as we are in a `BankAccount` class, so all `BankAccount` objects created within this class can access their **`private`** attributes. Now, whenever we need to copy a `BankAccount` object we just call the `clone` method. For example:

```
// create the original object
BankAccount ourAccount = new BankAccount ("98765432", "Charatan and Kans");
// now make a copy using the clone method, notice a type cast is required
BankAccount tempAccount = (BankAccount) ourAccount.clone();
// other instructions here
```

The `clone` method sends back an exact copy of the original account, not a copy of the reference (see Fig. 24.6).

Now, whatever we do to the copied object will leave the original object unaffected, and vice versa.

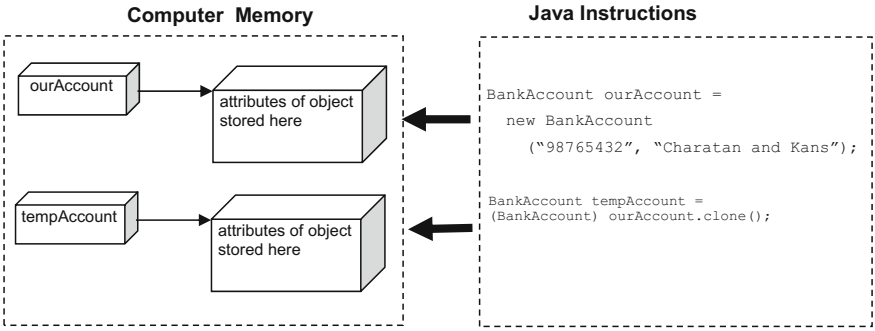


**Fig. 24.6** The *clone* method creates a copy of an object

In a similar way, we can ensure that classes that contain `BankAccount` objects do not inadvertently send back references (and hence aliases) to these objects:

```
public class Customer
{
    // as before here

    // next two methods now send back clones, not aliases
    public BankAccount getFirstAccount()
    {
        return (BankAccount)account1.clone();
    }

    public BankAccount getSecondAccount()
    {
        return (BankAccount)account2.clone();
    }
}
```

Now, referrring to our earlier example, the problem is removed because of the use of the `clone` method in the `Customer` class, as illustrated in the fragment below:

```
Customer someCustomer = new Customer();

// some code to update someCutomer here


/* now a temporary variable is created to read details of first account but this is not an alias
it is a clone */
BankAccount tempAccount = someCustomer.getFirstaccount();
System.out.println("balance of first account = " + tempAccount.getBalance();
// assume the balance is displayed as 500
temp.withdraw(100); /*  because temp is a clone the private BankAccount attribute
                        account1 is unaffected */
System.out.println("balance of first account = " +  someCustomer.getFirstAccount().getBalance());
// the balance of the customer's first account will be still be 500
```

### 24.3.3 Immutable Objects

We said that methods that return references to objects actually create aliases and that this can be dangerous. However, these aliases are not *always* dangerous. Consider the following features of the original `BankAccount` class:

```
public class BankAccount
{
    private String accountNumber;

    // other attributes and methods here

    public String getAccountNumber()
    {
        return accountNumber;
    }
}
```

In this case the `getAccountNumber` method returns a reference to a **`private`** `String` object (`accountNumber`). This is an alias for the **`private`** `String` attribute. However, this alias causes no harm as there are no `String` methods that allow a `String` object to be altered. So, this alias cannot be used to alter the **`private`** `String` object.

Objects which have no methods to alter their state are known as **immutable objects**. `String` objects are immutable objects. Objects of classes that you develop may also be immutable depending on the methods you have provided. If such objects are immutable, you do not have to worry about creating aliases of these objects and do not need to provide them with `clone` methods. For example, let's go back to the `Library` application (consisting of a collection of `Book` objects) that we developed in Chap. 15. Rather than showing you the code, Fig. 24.7 shows you the UML design for the `Library` and `Book` classes.

As you can see, the `Library` class contains a collection of `Book` objects. These `Book` objects are part of the **`private`** `books` attribute in the `Library` class. However the `getBook` method returns a reference to one of these `Book` objects and so sends back an alias. This is not a problem, however, because if you look at the design of the `Book` class the only methods provided are `get` methods.

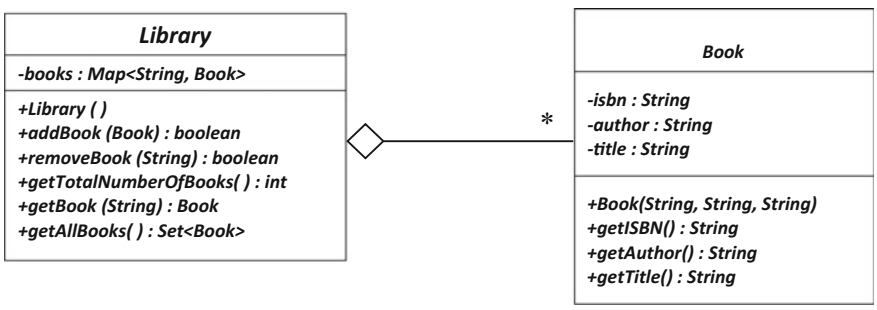


**Fig. 24.7** Design for the *Library* application

In other words, there are no Book methods that can alter the attributes of the Book object once the Book object has been created. A Book object is an immutable object.

### 24.3.4 Using the *clone* Method of the *Object* Class

Although the clone method in the Object class is not made available as part of your class's external interface, it can be used *within* classes that you develop. In particular, you might wish to use it within a clone method that you write yourself, as it does carry out the task of copying an object for you—albeit with some restrictions.

The clone method from the Object class copies the *memory contents* allocated to the object attributes. This is sometimes referred to as a *bit-wise copy*. This means that it makes exact copies of attributes that are of primitive type, and it makes copies of references for attributes that are objects. Of course, a copy of a reference gives you an alias—but if the object in question is immutable, this is not a problem. This means that:

- if a class's attributes are all of primitive type, then make your clone method just call the clone method of Object;
- if a class's attributes include objects, and these objects are *all* immutable, then again make your clone method just call the clone method of Object;
- if a class's attributes include any objects which are *not* immutable, then you *cannot* rely upon the clone method of Object to make a sensible copy and so you must write your own instructions for providing a clone.

Bearing these points in mind, let us revisit the clone method for our BankAccount class.

```
public class BankAccount
{
   // attributes
   private String accountNumber;
   private String accountName;
   private double balance;

   public Object clone()
   {
      // code goes here
   }

   // other code here
}
```

To make a copy of a BankAccount object we need a new bank account with an identical account number, name and balance. The balance attribute is of type **double**, so a bit-wise copy would be fine here. The account name and number are both String objects; since strings are immutable a bit-wise copy is fine here also. This means the *entire* object can be safely copied using the clone method of

`Object`. Here is a first attempt at using this method within our own `clone` method—it will not compile!:

```
// this attempt to clone a BankAccount will not compile
public Object clone()
{
   // call 'clone' method of superclass Object
   return super.clone();
}
```

This will not compile at the moment because the `clone` method of `Object` checks whether developers of this class really want to allow cloning to go ahead.

To indicate that developers do want cloning to go ahead, they have to mark their class as implementing the `Cloneable` interface. This interface, much like `Serializable`, contains no methods. It is just used to mark a class with some extra information. So, in order to call the `clone` method of `Object`, we need to mark the `BankAccount` class as follows:

```
// marking this class Cloneable allows us to call clone method of Object
public class BankAccount implements Cloneable
{
   // code here can use 'super.clone()'
}
```

There is one last thing we need to do in order to use the `clone` method of the `Object` class. This method throws a checked `CloneNotSupportedException` if the calling class does not implement the `Cloneable` interface. Of course we know our class does implement this interface, but as this is a checked exception, we still need to provide a **`try`**...**`catch`** around the call to `super.clone()` to keep the compiler happy. Here is the modified `BankAccount` class:

```
// mark that objects of this class can be cloned
public class BankAccount implements Cloneable
{
   // attributes as before
   private String accountNumber;
   private String accountName;
   private double balance;


   // this method allows BankAccount objects to be cloned
   public Object clone()
   {
      try
      {
         return super.clone(); // call 'clone' from Object
      }
      catch (CloneNotSupportedException e) // will never be thrown!
      {
         return null;
      }
   }

   // other code here
}
```

Whether or not you use `super.clone()` in your implementation of the `clone` method, it is always a good idea to mark your class `Cloneable`, so it is clear that objects from your class can be cloned.

### 24.3.5 Copy Constructors

The previous sections demonstrated how `clone` methods can be used to avoid aliases by providing exact (deep copies) of an object. But, as you could see, implementing `clone` methods can be a little tricky and using `clone` methods requires type-casting. A popular alternative to this approach is to provide copy constructors instead. Copy constructors provide a way of creating an exact copy of an object from an object sent as a parameter to a constructor. Languages like C++ automatically provide copy constructors, but in Java we have to implement them ourselves. Doing so is fairly straightforward. Let's return to the original `BankAccount` class and assume we have not included a `clone` method. Instead we will provide an additional copy constructor that receives a `BankAccount` object as a parameter and copies this parameter's attributes to make a new exact copy. Here is the outline of the class:

```
public class BankAccount
{
    // original attributes here

    // the original constructor
    public BankAccount(String numberIn, String nameIn)
    {
        accountNumber = numberIn;
        accountName = nameIn;
        balance = 0;
    }

    // the copy constructor
    public BankAccount(BankAccount accIn)
    {
        accountNumber = accIn.accountNumber;
        accountName = accIn.accountName;
        balance = accIn.balance;
    }

   // original methods here plus a toString method

}
```

You can see that, as well as the original constructor, we have provided a copy constructor that receives a `BankAccount` object and makes an exact copy by copying across every attribute value of the parameter object:

```
// the copy constructor
public BankAccount(BankAccount accIn)
{
    accountNumber = accIn.accountNumber; // copy account number
    accountName = accIn.accountName; // copy account name
    balance = accIn.balance; // copy account balance
}
```

The `CopyConstructorDemo` program below demonstrates how easy to is to use a copy constructor to create copies of objects. Note we are assuming a `toString` method has been included in the `BankAccount` class for ease of testing:

***CopyConstructorDemo***

```
public class CopyConstructorDemo
{
    public static void main(String[] args)
    {
        BankAccount b1 = new BankAccount ("001", "Justin Thyme"); // balance zero
        b1.deposit(100); // balance 100
        System.out.println("first object "+b1);
        BankAccount b2 = new BankAccount(b1); // create copy via copy constructor
        System.out.println("second object "+b2);// display copy
        b1.withdraw(50);// modify original object
        System.out.println("first object "+b1);
        System.out.println("second object "+b2);// second object untouched
    }
}
```

We have created a `BankAccount` object, `b1`, and deposited some funds into this object via the `deposit` method before displaying it (using its `toString` method). The next line is the key instruction where we use the copy constructor to create a new `BankAccount` object, `b2`, that is an exact copy of the first object:

```
BankAccount b2 = new BankAccount(b1); // create copy via copy constructor
```

You can see how simple this is. There is no need to type-cast as with a `clone` method. We then display the copy, before withdrawing money from the first object and displaying both objects again. Here is the program output:

```
first object (001, Justin Thyme, 100.0)
second object (001, Justin Thyme, 100.0)
first object (001, Justin Thyme, 50.0)
second object (001, Justin Thyme, 100.0)
```

As expected, the second object is an exact copy of the first object. Once the copy has been created we can modify the first object without modifying the copy.

Which technique you use for creating object copies (`clone` methods or copy constructors) is really up to you. You might find using the `Object` `clone` method simpler if the object in question has many attributes whereas a copy constructor may be easier for objects that have fewer attributes or attributes that are not able to be cloned simply (such as non-immutable objects).

### 24.3.6 Garbage Collection

When an object is created using the **`new`** operator, a request is being made to grab an area of free computer memory to store the object's attributes. Because this memory is requested during the running of a program, not during compilation, the compiler cannot guarantee that enough memory exists to meet this request. Memory could become exhausted for two related reasons:

- continual requests to grab memory are made when no more free memory exists;
- memory that is no longer needed is not released back to the system.

These problems are common to all programming languages and the danger of memory exhaustion is a real one for large programs, or programs running in a small memory space. Java allows both of the reasons listed above to be dealt with effectively and thus ensures that programs do not crash unexpectedly.

First, exception-handling techniques can be used to monitor for memory exhaustion and code can be written to ensure the program terminates gracefully. More importantly, Java has a built-in garbage collection facility to release unused memory. This is a facility that regularly trawls through memory looking for locations used by the program, freeing any locations that are no longer in use.

For example consider the program below.

***Tester***

```
import java.util.Scanner;

public class Tester
{
   public static void main(String[] args)
   {
      char ans;
      Scanner keyboard = new Scanner (System.in);
      Oblong object; // reference to object created here
      do
      {
         System.out.print("Enter length: ");
         double length = keyboard.nextDouble();
         System.out.print("Enter height: ");
         double height = keyboard.nextDouble();
         // new object created each time we go around the loop
         object = new Oblong(length, height);
         System.out.println("area = "+ object.calculateArea());
         System.out.println("perimeter = "+ object.calculatePerimeter());
         System.out.print("Do you want another go? ");
         ans = keyboard.next().charAt(0);
      }  while (ans == 'y' || ans == 'Y');
   }
}
```

Here, a new object is created each time we go around the loop. The memory used for the previous object is no longer required. In a language like C++ the memory occupied by old objects would not be destroyed unless the programmer added instructions to do so. So if the programmer forgot to do this, and this happened on a large scale in your C++ program, the available memory space could easily be exhausted. The Java system, however, regularly checks for such unused objects in memory and destroys them.

Although automatic garbage collection does make extra demands on the system (slowing it down while it takes place), this extra demand is considered by many to be worthwhile by removing a heavy burden on programmers. Nowadays many programming languages, such as C# and Python, also include a garbage collection facility.

**Table 24.1** TIOBE programming community index

| Position Aug 2018 | Position Aug 2017 | Programming language |
|---|---|---|
| 1 | 1 | Java |
| 2 | 2 | C |
| 3 | 3 | C++ |
| 4 | 5 | Python |
| 5 | 6 | Visual Basic.NET |
| 6 | 4 | C# |
| 7 | 7 | PHP |
| 8 | 8 | JavaScript |
| 9 | – | SQL |
| 10 | 14 | Assembly Language |

## 24.4 The Role of Java

While Java began life as a language aimed primarily at programming consumer devices, it has evolved into a sophisticated application programming language; competing with languages such as C++, Python and C#, to develop a wide range of applications. The security and reliability offered by the language has allowed the use of Java to be spread from desktop applications to network systems, web-based applications, set-top boxes, smart cards, computer games, smart phones and many more. To see an example of the enormous range of applications powered by Java visit the Oracle™ site at: http://go.java.

Table 24.1 gives the TIOBE programming community index of the ten most popular programming languages for August 2018.[1]

You can see that Java is at the top of this table, as it was last year. In fact it has been top of this index for many years.

## 24.5 What Next?

This chapter marks the end of our Java coverage for your second semester in programming. Although you have covered a lot of material, there is still more that you can explore. For example, we looked at how packages can be used to organise and distribute our Java applications in Chap. 19. The Java Programming Module System (JPMS) was introduced with the release of Java 9 and provides an even higher level of organisation to group together a collection of packages. We looked at Java applications that run over a local network in Chap. 23, but Java is also used

[1]The TIOBE index is a respected measure of the popularity of a programming language. For details of the table itself and of how it was compiled go to https://www.tiobe.com/tiobe-index/.

for the development of large distributed enterprise systems over wide area networks as well as cloud-based systems. We have also had a thorough look at JavaFX throughout this text but there is still much more you can find out about, including FXML—a Java FX tailored XML language developed by Oracle™. The good news is that you are now well placed to explore all these areas as well as many more. In the meantime, don't forget you can get further information on the Java language at the Oracle™ website https://www.oracle.com/java/.

Now that you have completed two semesters of programming we are pretty certain that you will have come to realize what an exciting and rewarding an activity it can be. So whether you are going on to a career in software engineering, or some other field in computing—or even if you are just going to enjoy programming for its own sake, we wish you the very best of luck for the future.

## 24.6 Self-test Questions

1. Distinguish between a *pointer* and a *reference*.
2. What does the term *multiple inheritance* mean and why does Java disallow it?
3. How do you implement a class that inherits two interfaces, both with a **`default`** method with the same name?
4. Consider the following class:

```
public class Critical
{
       private int value;

       public Critical (int valueIn)
       {
           value = valueIn;
       }
       public void setValue(int valueIn)
       {
           value = valueIn;
       }
       public int getValue ()
       {
           return value;
       }
}
```

(a) Explain why `Critical` objects are not *immutable*.

(b) Write fragments of code to create `Critical` objects and demonstrate the problem of *aliases*.

(c) Develop a `clone` method in the `Critical` class (make use of the `clone` method of `Object` here).

(d) Write fragments of code to demonstrate the use of this `clone` method.

(e) What is the purpose of a *copy constructor*?

(f) Develop a copy constructor for the `Critical` class.

(g) Write fragments of code to demonstrate the use of this copy constructor.

5. Look back at the classes from the two case studies of Chaps. 11, 12 and 21.

   (a) Which methods in these classes return aliases?

   (b) Which aliases could be dangerous?

   (c) How can these aliases be avoided?

6. What are the advantages and disadvantages of a *garbage collection* facility in a programming language?

## 24.7 Programming Exercises

1. Implement the `Critical` class of self-test question 4 and then write a tester program to demonstrate the problem of aliases.
2. Amend the `Critical` class by adding a `clone` method as discussed in self-test question 4(c) and then amend the tester program you developed in the previous programming exercise to demonstrate the use of this `clone` method.
3. Amend the `Critical` class further by adding the copy constructor discussed in self-test question 4(e) and then amend the tester program you developed in the previous programming exercise to demonstrate the use of this copy constructor.
4. Implement the changes you identified in self-test question 5, in order to remove the aliases that might have been present in the classes from the two case studies.
5. Review all the classes that you have developed so far and identify any problems with aliases. Use the techniques discussed in this chapter to avoid these aliases.