CS3231
Object Oriented Programming I

Name: ________________________________ (   ) Date: _______________

Chapter 4: Object Type

Lesson Objectives

After completing the lesson, the student will be able to

- *know the difference between a class and object*
- *understand the characteristics of an object*
- *create an object from a class*
- *use dot operator to access object members*
- *differentiate between primitive type and reference type*
- *use a reference type to access an object*
- *use the null reference in statements to preserve an object state*
- *understand garbage collection mechanism in Java*

4.1 Object Oriented Programming (OOP)

The two most important concepts in object-oriented programming are the **class** and the **object**. **A class can be regarded as an object template: it describes how an object looks and operates**. In the broadest term, an object is a thing, both tangible and intangible, that we can imagine. A program written in object-oriented style will consist of interacting objects.

Object-oriented programming (OOP) involves programming using objects. An object represents an entity in the real world that can be distinctly identified. For example, a student, a bicycle, a circle, a dog, a bank account, a button, and even a loan can all be viewed as objects. An object has a unique **identity**, **state**, and **behavior**.

An object is comprised of data and operations that manipulate these data. For example, a Student object may consist of data such as name, gender, birth date, home address, phone number and age, and operations for assigning and changing these data values.

- Software objects have **identity**. Each is a distinct chunk of memory. Each software object is a distinct entity even though it may look nearly the same as other objects of the same type. For example, every car should be given a unique engine serial number.

- The **state** of an object (also known as its properties or attributes) is represented by data fields with their **current values**. A circle object, for example, has a data field radius, which is the property that characterizes a circle. A rectangle object has the data fields such as width and height, which are the properties that characterize a rectangle. These values may change over time when an object interacts with other objects or perform some actions such as resizing.

- The **behavior** of an object (also known as its actions) is defined by methods. To invoke a method on an object is to ask the object to **perform an action**. For example, you may define methods named `getArea()` and `getPerimeter()` for circle objects. A circle object may invoke `getArea()` to return its area and `getPerimeter()` to return its perimeter. You may also define the `setRadius(radius)` method. A circle object can invoke this method to change its radius.

4.2 Difference Between Classes and Objects

Objects of the same type are defined using a common class. A class is a template, blueprint, or mould that defines what an object's data fields and methods will be. It tells the JVM how to make an object of the particular type. Each object made from that class can have its own values for the instance variables of that class.

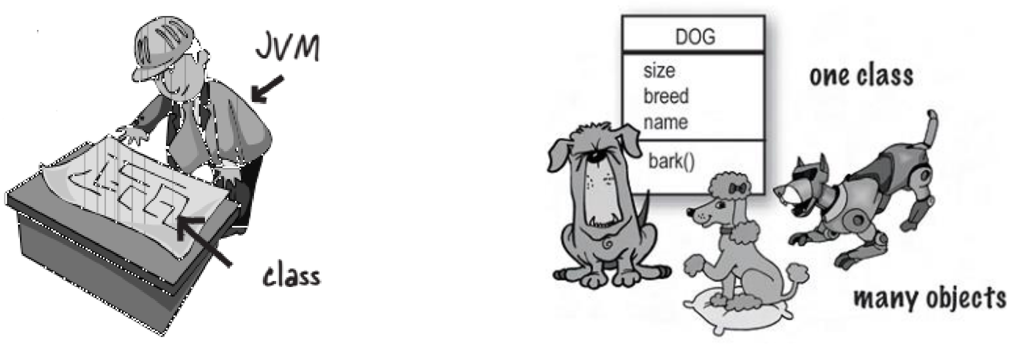


Figure 4.1 Class versus Objects.

An object is an instance of a class. See Figure 4.1. You can create many instances of a class. **Creating an instance is referred to as instantiation.** The terms object and instance are often interchangeable. The relationship between class and objects is analogous to that between a cookie-cutter and cookies. A class is not an object, but it is used to construct them. You can make as many cookies as you want from a single cookie-cutter.

Before you can create objects in a program to interact and perform tasks, you must first build classes to instantiate these objects. Figure 4.1 shows a Circle class and three objects created from the Circle class.

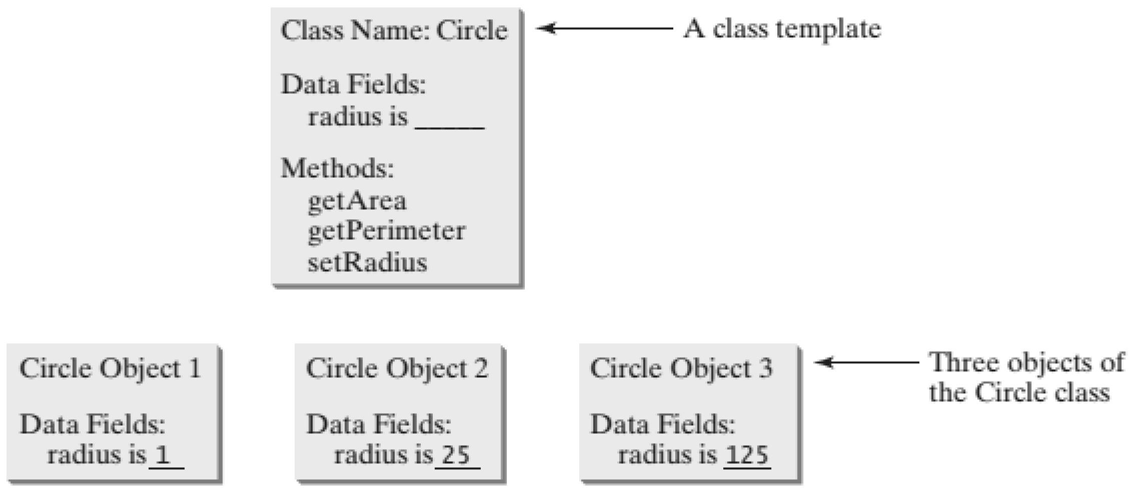


Figure 4.2 Shows a class named Circle and its three objects.

In a nutshell, the difference between a class and an object is:

- An object is a thing.
- A class is a design plan for things of that kind.

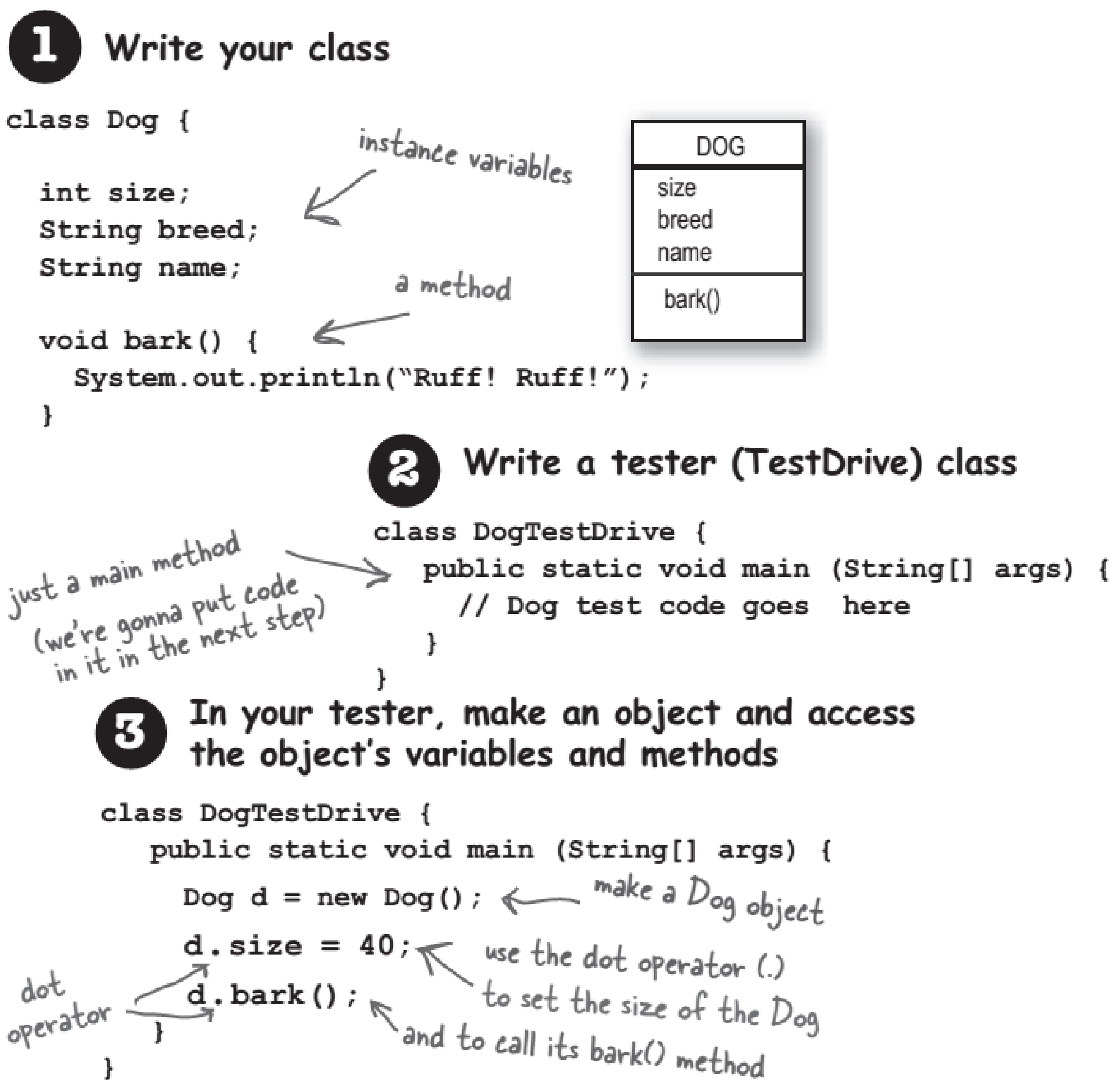
4.3 Creating Objects

To create and use objects, you need two classes. One class for the type of object you want to use (eg. Dog, Clock, Television, etc) and another class to test your new class. The tester class is where you put the `main()` method. **In the `main()` method, you create and access objects of your new class type**. The tester class (also known as **test driver**) has only one job: to try out the methods and variables of your new object class type.

In true object-oriented application, you will need to use more than one class in your program. Thus, a tester class will naturally become a focal point or platform for multiple types of objects to come together to interact. The two uses of `main()` method is to:

- test your class
- launch or start your Java application

Ideally, a class definition should only be doing what it is supposed to do. However, for the purpose of providing concise illustrations, the course material may sometimes include a class definition and a `main()` method in a single class. Figure 4.3 illustrate the steps to create `Dog` class, followed by a `Dog` object in the `DogTestDrive` class.


1 Write your class
instance variables
a method
DOG
size
breed
name
bark()
2 Write a tester (TestDrive) class
just a main method (we're gonna put code in it in the next step)
3 In your tester, make an object and access the object's variables and methods
make a Dog object
use the dot operator (.) to set the size of the Dog
and to call its bark() method
dot operator

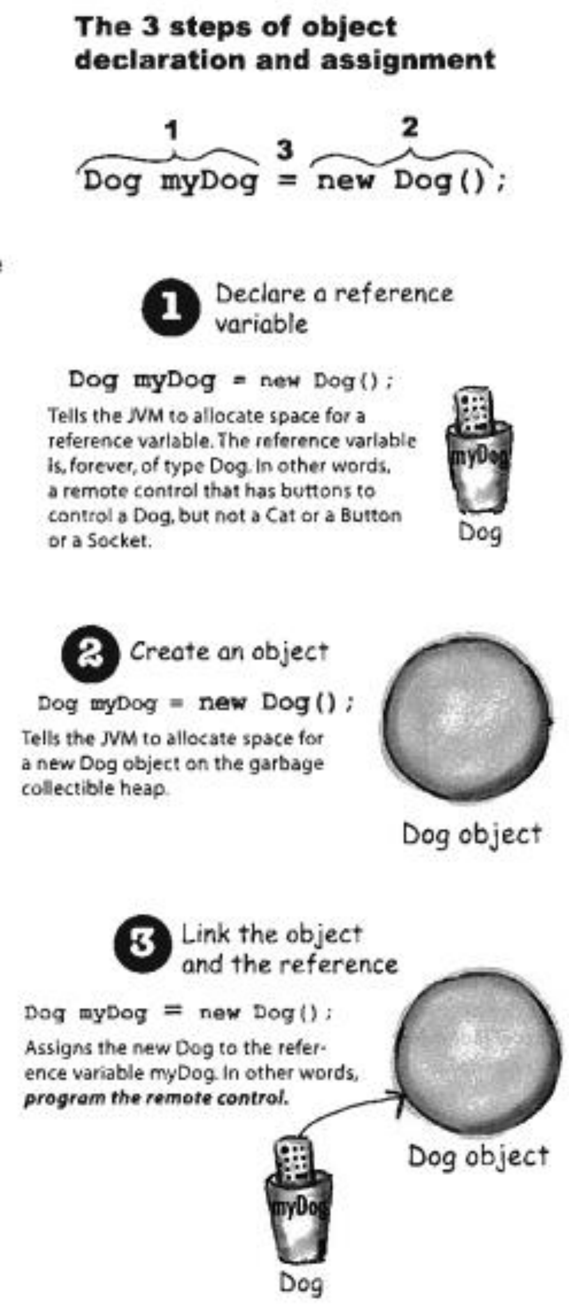
```
class Dog {

  int size;
  String breed;
  String name;

  void bark() {
    System.out.println("Ruff! Ruff!");
  }
```

```
class DogTestDrive {
   public static void main (String[] args) {
     // Dog test code goes  here
   }
}
```

```
class DogTestDrive {
   public static void main (String[] args) {
     Dog d = new Dog();
     d.size = 40;
     d.bark();
   }
}
```

Figure 4.3 Steps To Create An Object

The `new` keyword tells JVM to construct an object of the Dog type. After you have created an object, you use **dot operator (.)** to refer to the object's parts. The dot operator (.), also known as the **object member access operator** give you access to an object's state and behavior (variables and methods). You may think of the Dog reference variable as a remote controller. You use it to get the object `d` to do something (bark).



The 3 steps of object declaration and assignment

$$\overbrace{\texttt{Dog myDog}}^{1} \overset{3}{=} \overbrace{\texttt{new Dog()}}^{2}\texttt{;}$$

1 Declare a reference variable

```
Dog myDog = new Dog();
```

Tells the JVM to allocate space for a reference variable. The reference variable is, forever, of type Dog. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket.

2 Create an object

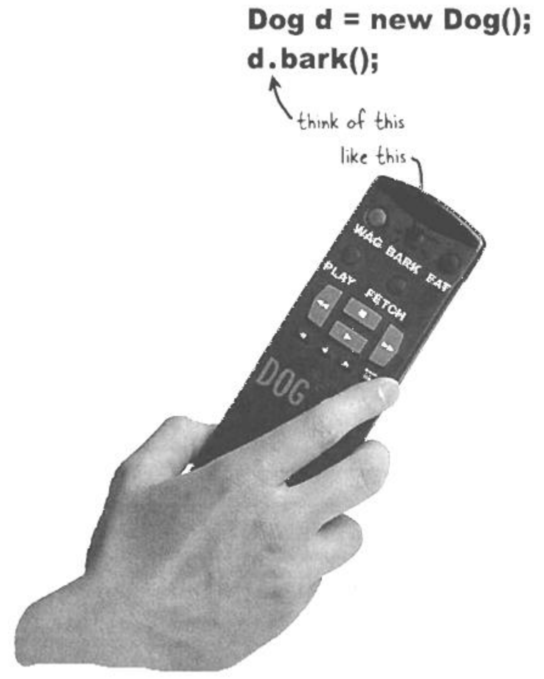
```
Dog myDog = new Dog();
```

Tells the JVM to allocate space for a new Dog object on the garbage collectible heap.

3 Link the object and the reference

```
Dog myDog = new Dog();
```

Assigns the new Dog to the reference variable myDog. In other words, ***program the remote control.***



```
Dog d = new Dog();
d.bark();
```

Figure 4.5 Reference is analogous to a remote controller.

Figure 4.4 Process of Declaring a Reference and Link an Object After Creation

4.4 Reference Type Versus Primitive Types

In an object-oriented language like Java, you create new, complex data types from simple primitives by creating a class. **Each class then serves as a new type in the language**. The word Dog stands for a brand-new type — a **object reference type (or simply reference type)** and the variable `d` reference an object of Dog type. Java has eight built-in primitive types and has as many reference types as people can define. Reference types and objects differ substantially from primitive types and their primitive values:

- **Eight primitive types are defined by the Java language, and the programmer cannot define new primitive types.** Reference types are user-defined, so there is an unlimited number of them. For example, a program might define a class named `Point` and use objects of this newly defined type to store and manipulate `x,y` points in a Cartesian coordinate system.

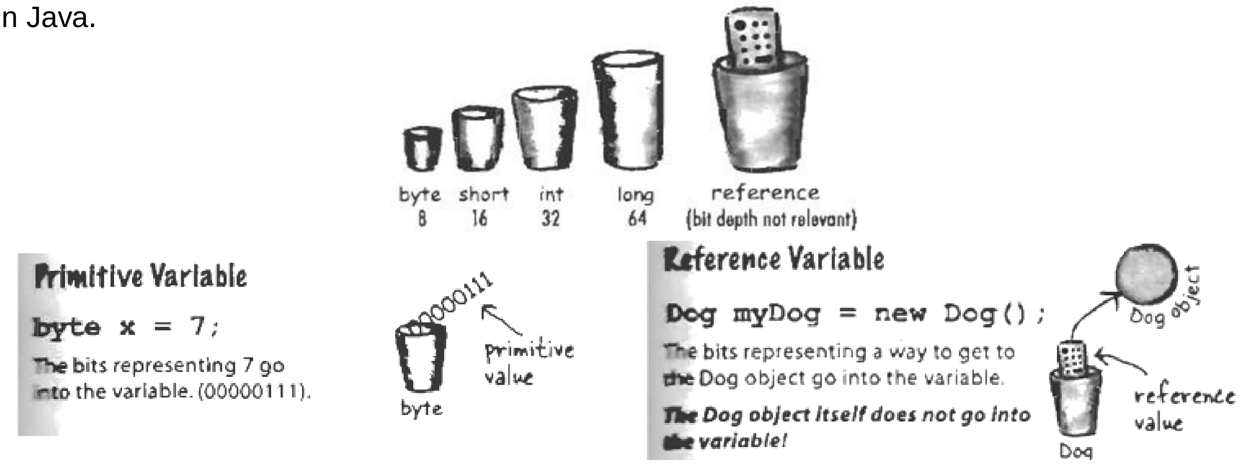
- **Primitive types represent single values. Reference types are aggregate types that hold zero or more primitive values or objects.** Our hypothetical `Point` class, for example, might hold two double values to represent the `x` and `y` coordinates of the points.

- Primitive types require between one and eight bytes of memory. All local variables (including method arguments) go on the stack memory. Variables allocated on the stack are stored directly to the memory and access to this memory is very fast, and its allocation is dealt with when the program is compiled. Objects, on the other hand, may require substantially more memory. Memory to store an object is dynamically allocated on the heap when the object is created and this memory is automatically "garbage collected" when the object is no longer needed.

When an object is assigned to a variable or passed to a method, the memory that represents the object is not copied. Instead, only a reference to that memory is stored in the variable or passed to the method.

Primitive Data	Objects
A primitive data value uses a small, fixed number of bytes.	An object is a big block of data. An object may use many bytes of memory.
There are only eight primitive data types. `byte` `short` `int` `long` `float` `double` `char` `boolean`	The data type of an object is called its **class**.
A programmer cannot create new primitive data types.	A programmer can invent new classes to meet the particular needs of a program.
A primitive can only hold a single value.	An object usually consists of many internal pieces which may include primitive type as well as reference type. Many classes are already defined in Java.

References are completely opaque in Java. If you are familiar with languages that use pointers, you will recognize a reference as another name for a pointer or a memory address. However, Java does not use the term pointer but instead uses the term reference. Moreover, these references are handled automatically. There are no programmer-accessible pointer (reference) operations for dereferencing or other pointer operations. The details are all handled automatically in Java.



```
byte x = 7;
```

The bits representing 7 go into the variable. (00000111).

```
Dog myDog = new Dog();
```

The bits representing a way to get to the Dog object go into the variable.

The Dog object itself does not go into the variable!

Figure 4.5 The difference between Reference Type and Primitive Type

4.5 Manipulating Objects and Reference Copies

Variables of a class type and variables of a primitive type behave quite differently in Java. Variables of a primitive type name their values in a straightforward way. For example, if `n` is an int variable, then `n` can contain a value of type `int`, such as 42. If `v` is a variable of a class type, then `v` does not directly contain an object of its class. Instead, `v` names an object by containing the memory address of where the object is located in memory.

The following code manipulates a primitive `int` value:

```
int x = 42;
int y = x;
```

After these lines execute, the variable `y` contains a copy of the value held in the variable `x`. Inside the JVM, there are two independent copies of the 32-bit integer 42.

However, what happens to a reference type is different from a primitive type. First we look at how an object reference is created and linked to the object. Figure 4.6 illustrates the State of Memory diagram for creating an object of Account class.

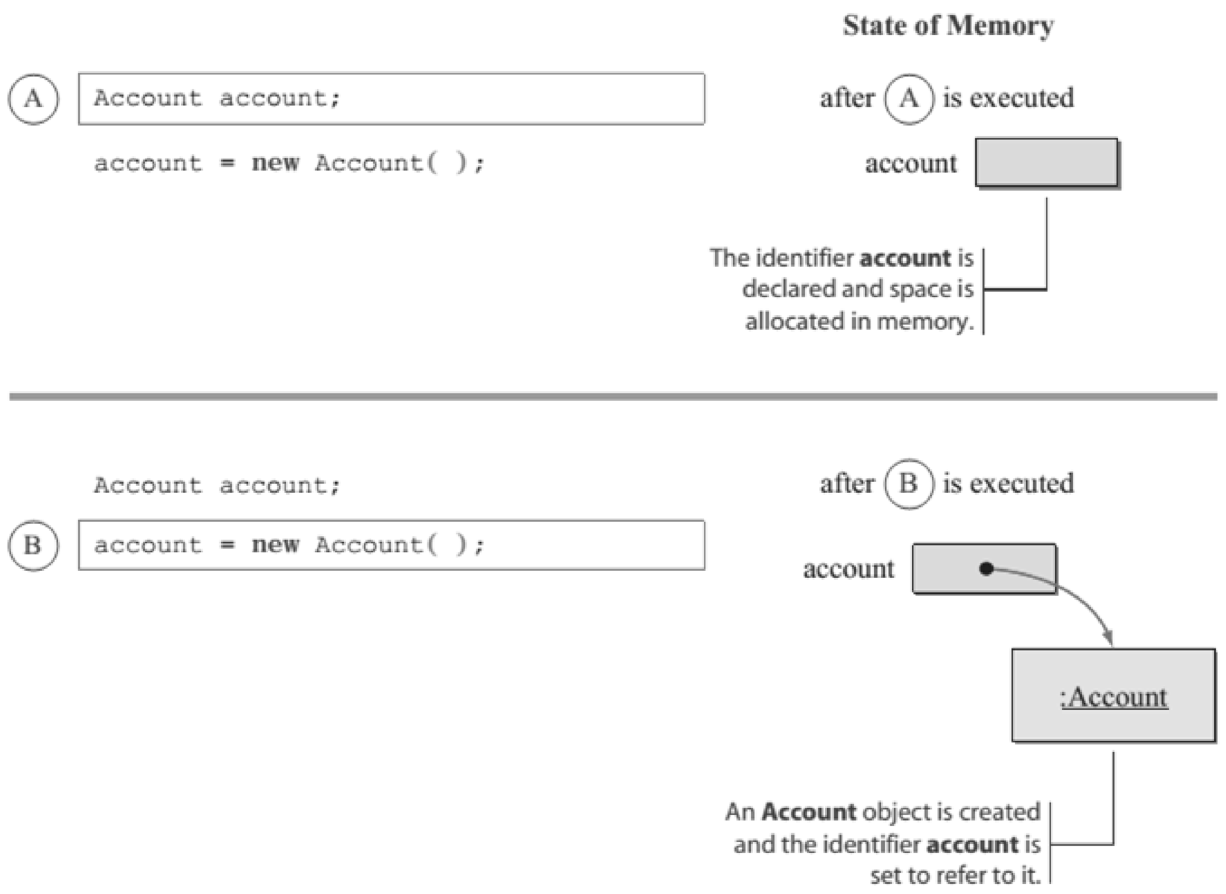

State of Memory
A
Account account;
account = new Account( );
after A is executed
account
The identifier account is declared and space is allocated in memory.
Account account;
B
account = new Account( );
after B is executed
account
:Account
An Account object is created and the identifier account is set to refer to it.

Figure 4.6 The State of Memory Diagram of Creating An Account Object

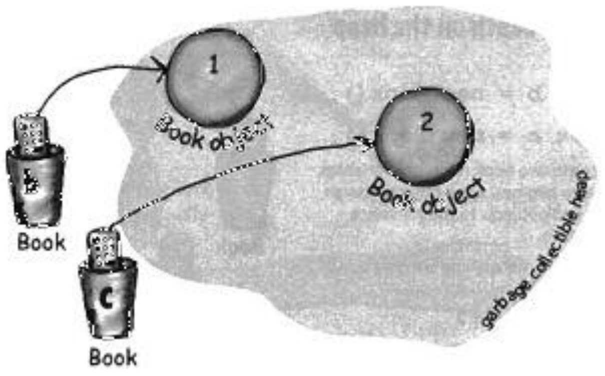
Assuming, you have a Book class. Now think about what happens if we run the same basic code but use a reference type instead of a primitive type. The following illustrates the state of memory when references of the same type are being assigned.

```
Book b = new Book();
Book c = new Book();
```

Declare two Book reference variables. Create two new Book objects. Assign the Book objects to the reference variables.

The two Book objects are now living on the heap. A heap is a memory area where objects are dynamically created and reside in.

References: 2
Objects: 2

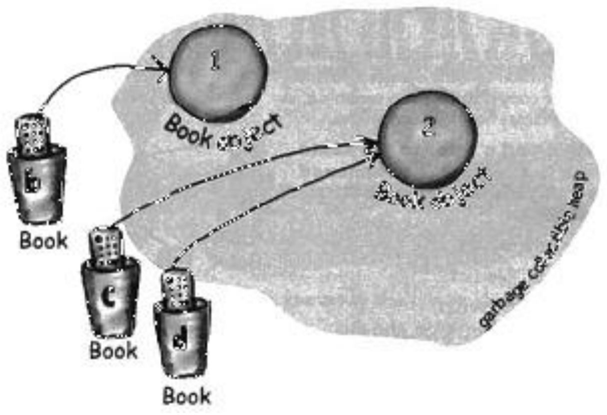


```
Book d = c;
```

Declare a new Book reference variable. Rather than creating a new third Book object, assign the value of the variable `c` to `d`. But what does it mean? It is saying take the content in `c` and makes a copy of them and put that copy (address) into `d`.

Both `c` and `d` refers to the same object. The `c` and `d` variables hold two different copies of the same value. Two remote controllers programmed to one Book object.

References: 3
Object: 2

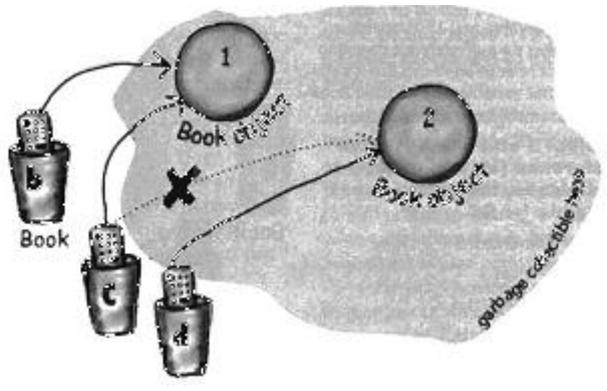


```
c = b;
```

Assign the value of variable `b` to variable `c`. By now you know what this means. The content inside variable `b` is copied, and that new copy is stuffed into variable `c`.

Both `b` and `c` refer to the same object.

References: 3
Object: 2

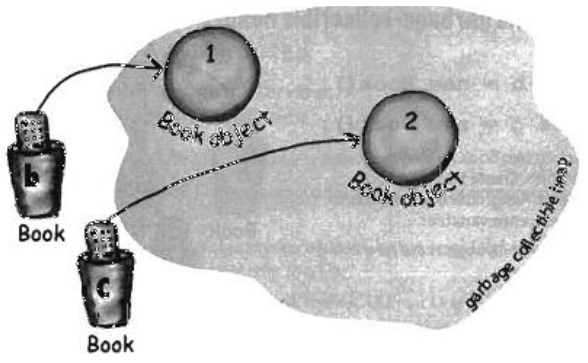


After this code runs, the variable `c` holds a copy of the reference held in the variable `b`. There is still only one copy of the Book object 2 in the JVM reference by `d`, but there are now two copies of the reference to that Book object 1. This has some important implications. If you modify Book object 1 through variable `c`, the object reference by `b` will also be affected because both variables hold references to the same object.

Another scenario that could happen is when an object variable currently linked to an object is being assigned a new reference. If not properly handled would result in case of an object being unreachable.

```
Book b = new Book();
Book c = new Book();
```

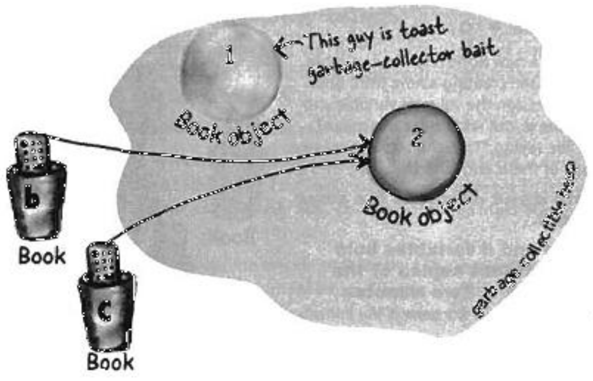
Declare two Book reference variables. Create two new Book objects. Assign the Book objects to the reference variables. The two book objects are now living on the heap.



Active References: 2
Reachable Objects: 2

```
Book b = c;
```

Assign the value of variable c to variable b. The content inside variable c is copied and that new copy is stuffed into variable b. Both variables hold identical values.



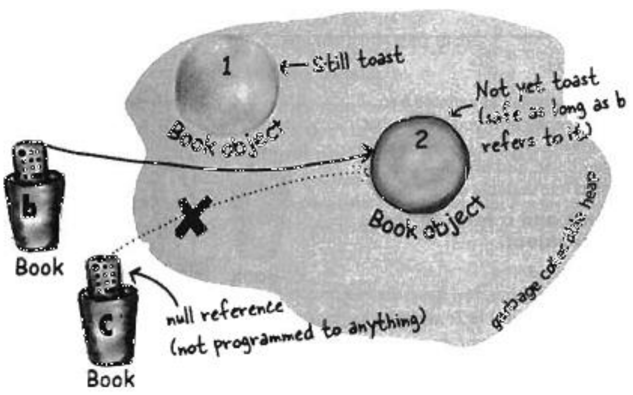
Both b and c refer to the same object. Object 1 is abandoned and eligible for Garbage Collection.

Active References: 2
Reachable Objects: 1
Abandoned Objects: 1

The first object that `b` referenced, Object 1 has no more references. It is unreachable and occupying memory space.

```
c = null;
```

Assign the value `null` to variable `c`. This makes `c` a `null` reference, meaning it does not refer to anything. But it is still a reference variable and another Book object can still be assign to it.



Object 2 still have active reference `b` and as long as it still does, the object is not eligible for Garbage Collection.

Active References: 1
null References: 1
Reachable Objects: 1
Abandoned Objects: 1

Remember that when an object is created, a certain amount of memory space is allocated for storing this object. If this allocated but unused space is not returned to the system for other uses, the space gets wasted. This returning of space to the system is called **deallocation**, and the mechanism to deallocate unused space is called **garbage collection**.

4.6 Garbage Collection

Java provides automatic memory management via garbage collection. Every object uses system resources, such as memory. We need a disciplined way to give resources back to the system when they ae no longer needed; otherwise, “resource leaks” might occur that would prevent resources from being reused by your program or possibly by other programs.

The JVM performs automatic garbage collection to reclaim the memory occupied by objects that are no longer used. When there are no more references to an object, the object is eligible to be collected. Collection typically occurs when the JVM executes its garbage collector, which may not happen for a while, or even at all before a program terminates. So, memory leaks that are common in other languages like C and C++ (because memory is not automatically reclaimed in those languages) are less likely in Java, but some can still happen in subtle ways. Resource leaks other than memory leaks can also occur. For example, an app may open a file on disk to modify its contents—if the app does not close the file, it must terminate before any other app can use the file.

4.7 Reference Data Fields and the `null` Value

The `null` keyword is a special literal value that is a reference to nothing, or an absence of a reference. The `null` value is unique because it is a member of every reference type. You can assign null to variables of any reference type. For example:

```
Point p = null;
Book b = null;
Dog d = null;
```

The data fields can be of reference types. For example, the following Student class contains a data field name of the String type. **String is a predefined Java class**.

```
class Student {
  String name; // name has the default value null
  int age; // age has the default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // gender has default value '\u0000'
}
```

If a data field of a reference type does not reference any object, the data field holds a special Java value, `null`.

The default value of a data field is `null` for a reference type, **0 for a numeric type, false for a boolean type, and \u0000 for a char type**. However, **Java assigns no default value to a local variable inside a method**. The following code displays the default values of the data fields `name`, `age`, `isScienceMajor`, and `gender` for a Student object:

```
class Test {
   public static void main(String[] args) {
   Student student = new Student();
   System.out.println("name? " + student.name);
   System.out.println("age? " + student.age);
   System.out.println("isScienceMajor? " + student.isScienceMajor);
   System.out.println("gender? " + student.gender);
 }
}
```

The following code has a compile error, because the local variables `x` and `y` are not initialized:

```
class Test {
    public static void main(String[] args) {
      int x; // x has no default value
      String y; // y has no default value
      System.out.println("x is " + x);
      System.out.println("y is " + y);
 }
}
```

NullPointerException is a common runtime error. It occurs when you invoke a method on a reference variable with a `null` value. Make sure you assign an object reference to the variable before invoking the method through the reference variable. Below code illustrates an example of a NullPointerException.

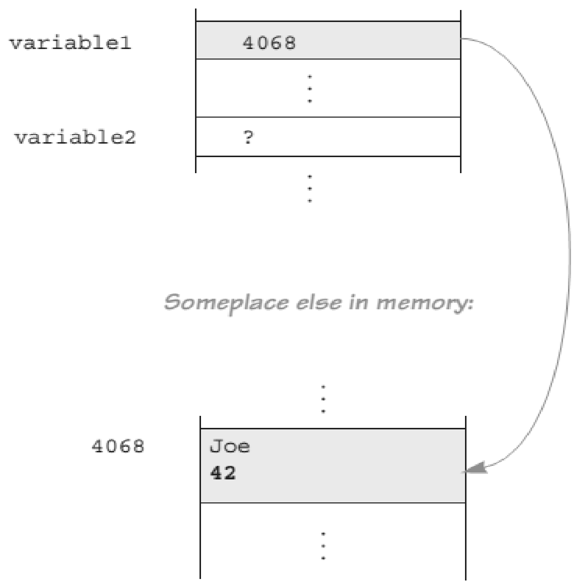
```
class Test {
    public static void main(String[] args) {
      String y = null; // y is a reference type of String with null value
      System.out.println(y.substring(0,5)); // cause a NullPointerException
 }
}
```

4.8 Further Example (Optional Reading)

Let us look at one more example. The ToyClass definition below has data members: `name` and `number`. The diagram below illustrates how reference type of a ToyClass behave in memory when they are being manipulated.

```
public class ToyClass {
   String name;
   int number;
   public ToyClass(String initialName, int initialNumber){
     name = initialName;
     number = initialNumber;
  }
}
```

```
ToyClass variable1 = new ToyClass("Joe", 42);
ToyClass variable2;
```



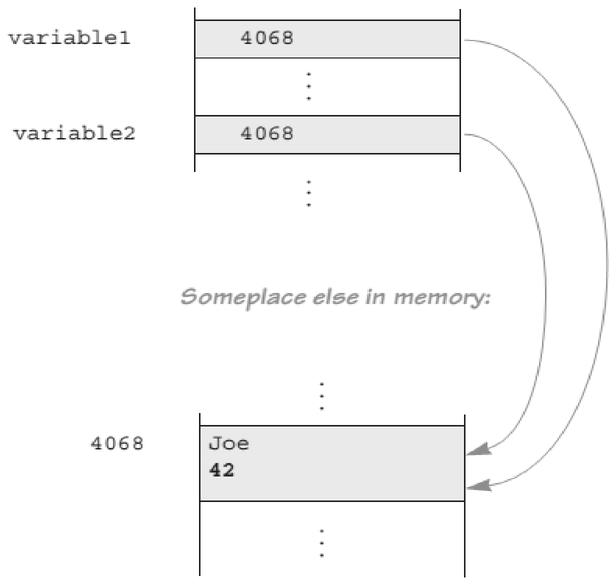
We do not know what memory address (reference) is stored in the variable `variable1`*. Let's say it is* `4068`*. The exact number does not matter.*

Note that you can think of

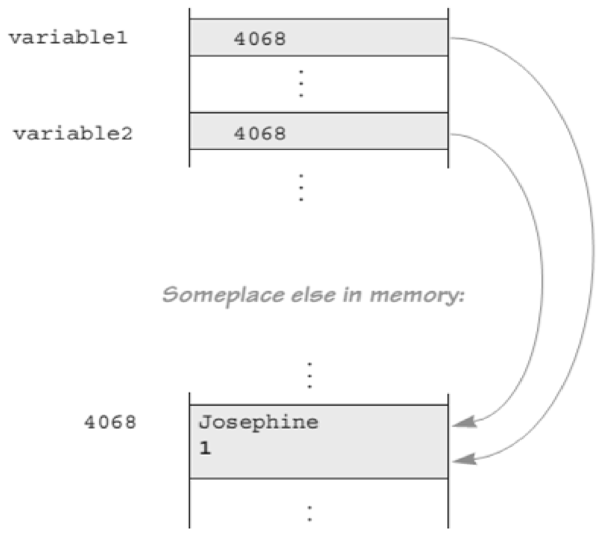
```
new ToyClass("Joe", 42)
```

as returning a reference.

```
variable2 = variable1;
```



```
variable2.set("Josephine", 1);
```



[Reference]

[1] Introduction to Java Programming Comprehensive Version 10th Ed, Daniel Liang, 2016.
[2] Java How To Program 10th Ed, Paul Deitel, Harvey Deitel, 2016.
[3] Beginning Programming With Java For Dummies, Barry Burn, Wiley 2014.
[4] Head First Java 2nd Ed, Katty Sierra, Bert Bates, O'Reilly.

[Self-Review Question]

1. Which operator is used to access a data field or invoke a method from an object?

2. What's the purpose of keyword new?

3. What is `NullPointerException`?

4. What is the output of the following code?

```
public class A {
  boolean x;
  public static void main(String[] args) {
        A a = new A();
         System.out.println(a.x);
  }

}
```