Classes and Objects

(Part 2, Objects and more)

Reading:

- Java Tutorial: Section *Classes and Objects* under section *Learning the Java Language*
- Textbook: Chapter 4 and 5
public class Point {
    public int x = 0;
    public int y = 0;
    // a constructor!
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}

Objects:
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
        origin = p;
    }
    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }
}
public class CreateObjectDemo {

    public static void main(String[] args) {

        // Declare and create a point object and two rectangle objects.
        Point originOne = new Point(23, 94);
        Rectangle rectOne = new Rectangle(originOne, 100, 200);
        Rectangle rectTwo = new Rectangle(50, 100);

        // display rectOne's width, height, and area
        System.out.println("Width of rectOne: " + rectOne.width);
        System.out.println("Height of rectOne: " + rectOne.height);
        System.out.println("Area of rectOne: " + rectOne.getArea());

        // set rectTwo's position
        rectTwo.origin = originOne;

        // display rectTwo's position
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);

        // move rectTwo and display its new position
        rectTwo.move(40, 72);
        System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
        System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);
    }
}

Here's the output:

    Width of rectOne: 100
    Height of rectOne: 200
    Area of rectOne: 20000
    X Position of rectTwo: 23
    Y Position of rectTwo: 94
    X Position of rectTwo: 40
    Y Position of rectTwo: 72
Life cycle of an Object:
- Creating objects
  - Declaring a variable to refer to an object
  - Instantiating an class
  - Initializing an object
- Using objects
  - Referencing a field
  - Calling a method
- Destroying objects (The garbage collector)

Creating Objects

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

Each of these statements has three parts:

1. **Declaration**: The code set in red are all variable declarations that associate a variable name with an object type.
2. **Instantiation**: The `new` keyword is a Java operator that creates the object.
3. **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object.
Declaring a Variable to Refer to an Object

Previously, you learned that to declare a variable, you write:

\[ \text{type name;} \]

This notifies the compiler that you will use \textit{name} to refer to data whose type is \textit{type}. With a primitive variable, this declaration also reserves the proper amount of memory for the variable.

You can also declare a reference variable on its own line. For example:

\textbf{Point originOne;}

\textbf{Does not create an object! Only creates a reference with no pointer...}

Instantiating a Class

The \textbf{new} operator instantiates a class by allocating memory for a new object and returning a reference to that memory.

The \textbf{new} operator also invokes the object constructor.
Note: The phrase "instantiating a class" means the same thing as "creating an object." When you create an object, you are creating an "instance" of a class, therefore "instantiating" a class.

The `new` operator requires a single, postfix argument: a call to a constructor. The name of the constructor provides the name of the class to instantiate.

The `new` operator returns a reference to the object it created. This reference is usually assigned to a variable of the appropriate type, like:

```java
Point originOne = new Point(23, 94);
```

The reference returned by the `new` operator does not have to be assigned to a variable. It can also be used directly in an expression. For example:

```java
int height = new Rectangle().height;
```
Initializing an Object

Here's the code for the `Point` class:

```java
public class Point {
    public int x = 0;
    public int y = 0;
    // constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

This class contains a single `constructor`. How can you recognize a constructor?

```java
Point originOne = new Point(23, 94);
```

The result of executing this statement can be illustrated in the next figure:
Here's the code for the `Rectangle` class, which contains four constructors:

```java
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
        origin = p;
    }
    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }
}
```
Each constructor lets you provide initial values for the rectangle's origin, width, and height, using both primitive and reference types.

If a class has multiple constructors, they must have different signatures. The Java compiler differentiates the constructors based on the number and the type of the arguments.

When the Java compiler encounters the following code, it knows to call the constructor in the `Rectangle` class that requires a `Point` argument followed by two integer arguments:

```java
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

An object can have multiple references to it.
Rectangle rectTwo = new Rectangle(50, 100);

A no-argument constructor:

Rectangle rect = new Rectangle();

All classes have at least one constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, called the default constructor. This default constructor calls the class parent's no-argument constructor, or the Object constructor if the class has no other parent. If the parent has no constructor (Object does have one), the compiler will reject the program.

Using Objects

Once you've created an object, you probably want to use it for something...
Referencing an Object's Fields

Object fields are accessed by their name. You must use a name that is unambiguous.

You may use a simple name for a field within its own class. For example, we can add a statement within the Rectangle class that prints the width and height:

```
System.out.println("Width and height are: " + width + ", " + height);
```

Code that is outside the object's class must use an object reference or expression, followed by the dot (.) operator, followed by a simple field name, as in:

```
objectReference.fieldName
```

For example:

```
System.out.println("Width of rectOne: " + rectOne.width);
System.out.println("Height of rectOne: " + rectOne.height);
```

Attempting to use the simple names width and height from the code in the CreateObjectDemo class doesn't make sense — those fields exist only within an object — and results in a compiler error.
When you access an instance field through an object reference, you reference that particular object's field. The two objects `rectOne` and `rectTwo` in the `CreateObjectDemo` program have different `origin`, `width`, and `height` fields.

**Unreferenced/Anonymous object?**

```java
int height = new Rectangle().height;
```

This statement creates a new `Rectangle` object and immediately gets its height.

Note that after this statement has been executed, the program no longer has a reference to the created `Rectangle`, because the program never stored the reference anywhere.

The object is unreferenced, and its resources are free to be recycled by the Java Virtual Machine.
Calling an Object's Methods

objectReference.methodName(argumentList);

or:

objectReference.methodName();

The Rectangle class has two methods: getArea() to compute the rectangle's area and move() to change the rectangle's origin. Here's the CreateObjectDemo code that invokes these two methods:

```java
System.out.println("Area of rectOne: " + rectOne.getArea());
...
rectTwo.move(40, 72);
```

The new operator returns an object reference, so you can use the value returned from new to invoke a new object's methods:

```java
new Rectangle(100, 50).getArea()
```

The expression new Rectangle(100, 50) returns an object reference that refers to a Rectangle object. As shown, you can use the dot notation to invoke the new Rectangle's getArea() method to compute the area of the new rectangle.
int areaOfRectangle = new Rectangle(100, 50).getArea();

Remember, invoking a method on a particular object is the same as sending a message to that object. In this case, the object that getArea() is invoked on is the rectangle returned by the constructor.

The Garbage Collector (Object destruction)

Managing memory explicitly is tedious and error-prone.

The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called garbage collection.

An object is eligible for garbage collection when there are no more references to that object.

References that are held in a variable are usually dropped when the variable goes out of scope.

Or, you can explicitly drop an object reference by setting the variable to the special value null. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a garbage collector that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically when it determines that the time is right.
More on Classes

- Returning values from methods.
- The `this` keyword.
- Class vs. instance members.
- Access control.

Returning a Value from a Method

A method returns to the code that invoked it when it

- completes all the statements in the method,
- reaches a `return` statement, or
- throws an exception (covered later),

Whichever occurs first.

You declare a method's return type in its method declaration. Within the body of the method, you use the `return` statement to return the value.

Any method declared `void` doesn't return a value. It does not need to contain a `return` statement, but it may do so. In such a case, a `return` statement can be used to branch out of a control flow block and exit the method and is simply used like this:

```java
return;
```
Any method that is not declared `void` must contain a `return` statement with a corresponding return value, like this:

```
return returnValue;
```

The data type of the return value must match the method's declared return type.

```
// a method for computing the area of the rectangle
public int getArea() {
    return width * height;
}
```

**A method can also return a reference type.**

```
public Bicycle seeWhosFastest(Bicycle myBike, Bicycle yourBike, Environment env) {
    Bicycle fastest;
    // code to calculate which bike is
    // faster, given each bike's gear
    // and cadence and given the
    // environment (terrain and wind)
    return fastest;
}
```

**Returning a Class or Interface**

When a method uses a class name as its return type, such as `seeWhosFastest` does, the class of the type of the returned object must be either a subclass of, or the exact class of, the return type.
Suppose that you have a class hierarchy in which `ImaginaryNumber` is a subclass of `java.lang.Number`, which is in turn a subclass of `Object`, as illustrated in the following figure.

![Class Hierarchy](image)

The class hierarchy for `ImaginaryNumber`

Now suppose that you have a method declared to return a `Number`:

```java
public Number returnANumber() {
    ...
}
```

The `returnANumber` method can return an `ImaginaryNumber` but not an `Object`. `ImaginaryNumber` is a `Number` because it's a subclass of `Number`. However, an `Object` is not necessarily a `Number` — it could be a `String` or another type.
You can override a method and define it to return a subclass of the original method, like this:

```java
public ImaginaryNumber returnANumber() {
    ...
}
```

This technique, called **covariant return type**, means that the return type is allowed to vary in the same direction as the subclass.

*(Skip this now. We shall return here after finishing the lesson on inheritance)*

---

**Note:** You also can use interface names as return types. In this case, the object returned must implement the specified interface.
Using the this Keyword

Within an instance method or a constructor, this is a reference to the current object — the object whose method or constructor is being called.

You can refer to any member of the current object from within an instance method or a constructor by using this.

Using this with a Field

```java
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

But it could have been written like this:

```java
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```
Each argument to the constructor shadows one of the object's fields — inside the constructor \( x \) is a local copy of the constructor's first argument. To refer to the Point field \( x \), the constructor must use \texttt{this.x}.

**Using this with a Constructor**

From within a constructor, you can also use the \texttt{this} keyword to call another constructor in the same class. Doing so is called an *explicit constructor invocation*.

```java
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 1, 1);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

If present, the invocation of another constructor must be the first line in the constructor.
Copy Constructor (see textbook)

A copy constructor is a constructor with one parameter of the same type as the class.

A copy constructor should be designed so the object it creates is intuitively an exact copy of its parameter.

```java
public final class Galaxy {

    private double fMass;
    private final String fName;

    //Regular constructor
    public Galaxy(double aMass, String aName) {
        fMass = aMass;
        fName = aName;
    }

    //Copy constructor
    public Galaxy(Galaxy aGalaxy) {
        this(aGalaxy.getMass(), aGalaxy.getName());
        //no defensive copies are created here, since
        //there are no mutable object fields(String is immutable)
    }

    //Alternative style for a copy constructor, using a static newInstance
    //What is this? A static method, not a constructor
    public static Galaxy newInstance(Galaxy aGalaxy) {
        return new Galaxy(aGalaxy.getMass(), aGalaxy.getName());
    }
    ...
}
```
Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- **At the top level**—public, or package-private (no explicit modifier).
- **At the member level**—public, private, protected, or package-private (no explicit modifier).

If a class has no modifier (the default, also known as package-private), it is visible only within its own package.

The following table shows the access to members permitted by each modifier.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>no modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Classes and Packages of the Example Used to Illustrate Access Levels
The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

<table>
<thead>
<tr>
<th>Visibility</th>
<th>Modifier</th>
<th>Alpha</th>
<th>Beta</th>
<th>AlphaSub</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>no modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

**Tips on Choosing an Access Level:**

If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

- Use the most restrictive access level that makes sense for a particular member. Use `private` unless you have a good reason not to.
- Avoid `public` fields except for constants. (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.) Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.
Understanding Class Members

We use the `static` keyword to create fields and methods that belong to the class, rather than to an instance of the class.

**Class Variables**

Sometimes, you want to have variables that are common to all objects.

Fields that have the `static` modifier in their declaration are called `static fields` or `class variables`.

They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory.
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    // add an instance variable for the object ID
    private int id;

    // add a class variable for the
    // number of Bicycle objects instantiated
    private static int numberOfBicycles = 0;

    ...
}

Class variables are referenced by the class name itself, as in

Bicycle.numberOfBicycles

This makes it clear that they are class variables.

Note: You can also refer to static fields with an object reference like

myBike.numberOfBicycles

But this is discouraged because it does not make it clear that they are class variables.
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;

        // increment number of Bicycles
        // and assign ID number
        id = ++numberOfBicycles;
    }

    // new method to return the ID instance variable
    public int getID() {
        return id;
    }

    ...
}

Class Methods

Static methods, which have the static modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class, as in

ClassName.methodName(args)
Note: You can also refer to static methods with an object reference like
instanceName.methodName(args)
but this is discouraged because it does not make it clear that they are class methods.

A common use for static methods is to access static fields.

```java
public static int getNumberOfBicycles() {
    return numberOfBicycles;
}
```

Not all combinations of instance and class variables and methods are allowed:

- Instance methods can access instance variables and instance methods directly.
- Instance methods can access class variables and class methods directly.
- Class methods can access class variables and class methods directly.
- Class methods **cannot** access instance variables or instance methods directly—they must use an object reference. Also, class methods cannot use the `this` keyword as there is no instance for `this` to refer to.
Constants

The static modifier, in combination with the final modifier, is also used to define constants. The final modifier indicates that the value of this field cannot change.

```
static final double PI = 3.141592653589793;
```

Constants defined in this way cannot be reassigned, and it is a compile-time error if your program tries to do so.

By convention, the names of constant values are spelled in uppercase letters. If the name is composed of more than one word, the words are separated by an underscore (_).

---

**Note:** If a primitive type or a string is defined as a constant and the value is known at compile time, the compiler replaces the constant name everywhere in the code with its value. This is called a compile-time constant. If the value of the constant in the outside world changes (for example, if it is legislated that pi actually should be 3.975), you will need to recompile any classes that use this constant to get the current value.
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    private int id;

    private static int numberOfBicycles = 0;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;

        id = ++numberOfBicycles;
    }

    public int getID() {
        return id;
    }

    public static int getNumberOfBicycles() {
        return numberOfBicycles;
    }

    public int getCadence() {
        return cadence;
    }

    public void setCadence(int newValue) {
        cadence = newValue;
    }
}
public int getGear(){
    return gear;
}

public void setGear(int newValue){
    gear = newValue;
}

public int getSpeed(){
    return speed;
}

public void applyBrake(int decrement){
    speed -= decrement;
}

public void speedUp(int increment){
    speed += increment;
}


Initializing Fields

As you have seen, you can often provide an initial value for a field in its declaration:

public class BedAndBreakfast {

    // initialize to 10
    public static int capacity = 10;

    // initialize to false
    private boolean full = false;
}


This works well when the initialization value is available and the initialization can be put on one line. However, this form of initialization has limitations because of its simplicity.

If initialization requires some logic (for example, error handling or a for loop to fill a complex array), simple assignment is inadequate.

Instance variables can be initialized in constructors, where error handling or other logic can be used. To provide the same capability for class variables, the Java programming language includes static initialization blocks.

Note: It is not necessary to declare fields at the beginning of the class definition, although this is the most common practice. It is only necessary that they be declared and initialized before they are used.

Static Initialization Blocks

A static initialization block is a normal block of code enclosed in braces, { }, and preceded by the static keyword. Here is an example:

```java
static {
    // whatever code is needed for initialization goes here
}
```

A class can have any number of static initialization blocks, and they can appear anywhere in the class body. The runtime system guarantees that static initialization blocks are called in the order that they appear in the source code.
There is an alternative to static blocks — you can write a private static method:

class Whatever {
    public static varType myVar = initializeClassVariable();

    private static varType initializeClassVariable() {
        // initialization code goes here
    }
}

The advantage of private static methods is that they can be reused later if you need to reinitialize the class variable.

**Initializing Instance Members**

Normally, you would put code to initialize an instance variable in a constructor. There are two alternatives to using a constructor to initialize instance variables: initializer blocks and final methods.

Initializer blocks for instance variables look just like static initializer blocks, but without the `static` keyword:

```java
{
    // whatever code is needed for initialization goes here
}
```

The Java compiler copies initializer blocks into every constructor. Therefore, this approach can be used to share a block of code between multiple constructors.
A final method cannot be overridden in a subclass. This is discussed in the lesson on interfaces and inheritance. Here is an example of using a final method for initializing an instance variable:

```java
class Whatever {
    private varType myVar = initializeInstanceVariable();

    protected final varType initializeInstanceVariable() {
        // initialization code goes here
    }
}
```

This is especially useful if subclasses might want to reuse the initialization method. The method is final because calling non-final methods during instance initialization can cause problems.

We shall return here later.