Java Programming Tutorial
Object-oriented Programming (OOP) Basics

1. Why OOP?

Suppose that you want to assemble your own PC, you go to a hardware store and pick up a motherboard, a processor, some RAMs, a hard disk, a casing, a power supply, and put them together. You turn on the power, and the PC runs. You need not worry whether the CPU is 1-core or 6-core; the motherboard is a 4-layer or 6-layer; the hard disk has 4 plates or 6 plates, 3 inches or 5 inches in diameter; the RAM is made in Japan or Korea, and so on. You simply put the hardware *components* together and expect the machine to run. Of course, you have to make sure that you have the correct *interfaces*, i.e., you pick an IDE hard disk rather than a SCSI hard disk, if your motherboard supports only IDE; you have to select RAMs with the correct speed rating, and so on. Nevertheless, it is not difficult to set up a machine from hardware *components*.

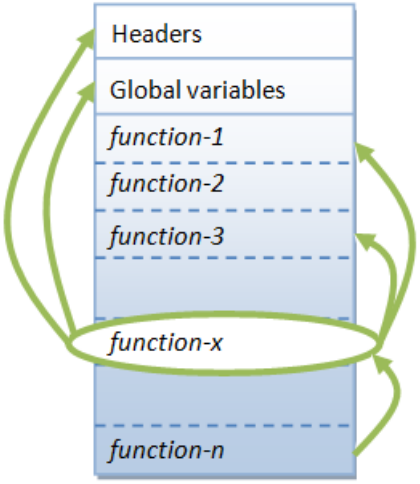
Similarly, a car is assembled from parts and components, such as chassis, doors, engine, wheels, brake and transmission. The components are reusable, e.g., a wheel can be used in many cars (of the same specifications).

Hardware, such as computers and cars, are assembled from parts, which are *reusable hardware components*.

How about software? Can you "assemble" a software application by picking a routine here, a routine there, and expect the program to run? The answer is obviously NO! Unlike hardware, it is very difficult to "assemble" an application from *software components*. Since the advent of computer 70 years ago, we have written tons and tons of programs and routines. However, for each new application, we have to re-invent the wheels and write the program from scratch!

Why re-invent the wheels? Why re-writing codes? Can you write better codes than those codes written by the experts?

Traditional Procedural-Oriented languages



A function (in C) is not well-encapsulated

Traditional procedural-oriented programming languages (such as C, Fortran, Cobol and Pascal) suffer some notable drawbacks in creating *reusable software components*:

1. The procedural-oriented programs are made up of functions. Functions are *less reusable*. It is very difficult to copy a function from one program and reuse in another program because the function is likely to reference the global variables and other functions. In other words, functions are not well-encapsulated as a self-contained *reusable unit*.
2. The procedural languages are not suitable of *high-level abstraction* for solving real life problems. For example, C programs uses constructs such as if-else, for-loop, array, method, pointer, which are low-level and hard to abstract real problems such as a Customer Relationship Management (CRM) system or a computer soccer game.

The traditional procedural-languages *separate* the data structures (variables) and algorithms (functions).

In the early 1970s, the US Department of Defense (DoD) commissioned a task force to investigate why its IT budget always went out of control; but without much to show for. The findings are:

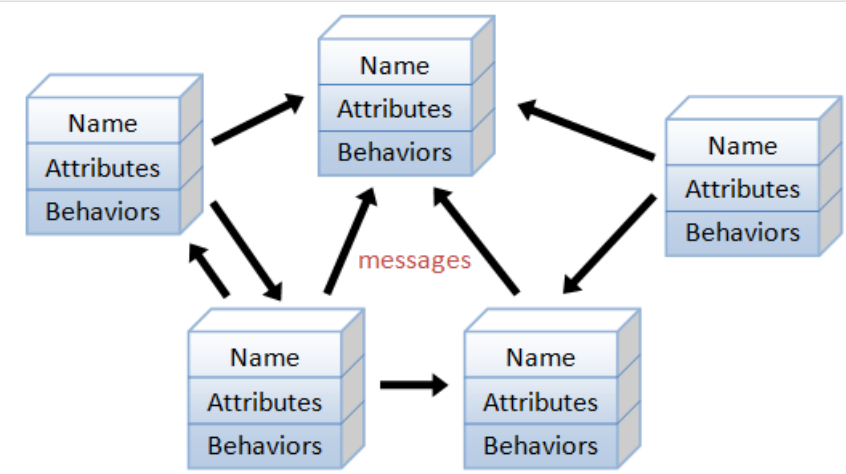
1. 80% of the budget went to the software (with the remaining 20% to the hardware).
2. More than 80% of the software budget went to maintenance (only the remaining 20% for new software development).

3. Hardware components could be applied to various products, and their integrity normally did not affect other products. (Hardware can share and reuse! Hardware faults are isolated!)
4. Software procedures were often non-sharable and not reusable. Software faults could affect other programs running in computers.

The task force proposed to make software behave like hardware OBJECT. Subsequently, DoD replaces over 450 computer languages, which were then used to build DoD systems, with an object-oriented language called Ada.

Object-Oriented Programming Languages

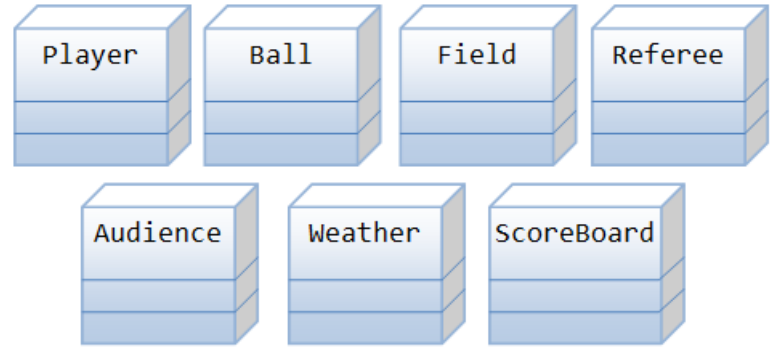
Object-oriented programming (OOP) languages are designed to overcome these problems.



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

1. The basic unit of OOP is a *class*, which encapsulates both the *static properties* and *dynamic operations* within a "box", and specifies the public interface for using these boxes. Since classes are well-encapsulated, it is easier to reuse these classes. In other words, OOP combines the data structures and algorithms of a software entity inside the same box.
2. OOP languages permit *higher level of abstraction* for solving real-life problems. The traditional procedural language (such as C and Pascal) forces you to think in terms of the structure of the computer (e.g. memory bits and bytes, array, decision, loop) rather than thinking in terms of the problem you are trying to solve. The OOP languages (such as Java, C++ and C#) let you think in the problem space, and use software objects to represent and abstract entities of the problem space to solve the problem.

As an example, suppose you wish to write a computer soccer games (which I consider as a complex application). It is quite difficult to model the game in procedural-oriented languages. But using OOP languages, you can easily model the program accordingly to the "real things" appear in the soccer games.



Classes (Entities) in a Computer Soccer Game

- Player: attributes include name, number, location in the field, and etc; operations include run, jump, kick-the-ball, and etc.
- Ball:
- Reference:
- Field:
- Audience:
- Weather:

Most importantly, some of these classes (such as `Ball` and `Audience`) can be reused in another application, e.g., computer basketball game, with little or no modification.

Benefits of OOP

The procedural-oriented languages focus on procedures, with function as the basic unit. You need to first figure out all the functions and then think about how to represent data.

The object-oriented languages focus on components that the user perceives, with objects as the basic unit. You figure out all the objects by putting all the data and operations that describe the user's interaction with the data.

Object-Oriented technology has many benefits:

- *Ease in software design* as you could think in the problem space rather than the machine's bits and bytes. You are dealing with high-level concepts and abstractions. Ease in design leads to more productive software development.
- *Ease in software maintenance*: object-oriented software are easier to understand, therefore easier to test, debug, and maintain.
- *Reusable software*: you don't need to keep re-inventing the wheels and re-write the same functions for different situations. The fastest and safest way of developing a new application is to reuse existing codes - fully tested and proven codes.

2. OOP in Java

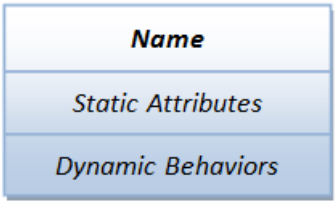
2.1 Class & Instances

In Java, *a class is a definition of objects of the same kind*. In other words, a *class* is a blueprint, template, or prototype that defines and describes the *static attributes* and *dynamic behaviors* common to all objects of the same kind.

An *instance* is *a realization of a particular item of a class*. In other words, an instance is an *instantiation* of a class. All the instances of a class have similar properties, as described in the class definition. For example, you can define a class called "`Student`" and create three instances of the class "`Student`" for "`Peter`", "`Paul`" and "`Pauline`".

The term "*object*" usually refers to *instance*. But it is often used loosely, and may refer to a class or an instance.

2.2 A Class is a 3-Compartment Box Encapsulating Data and Operations



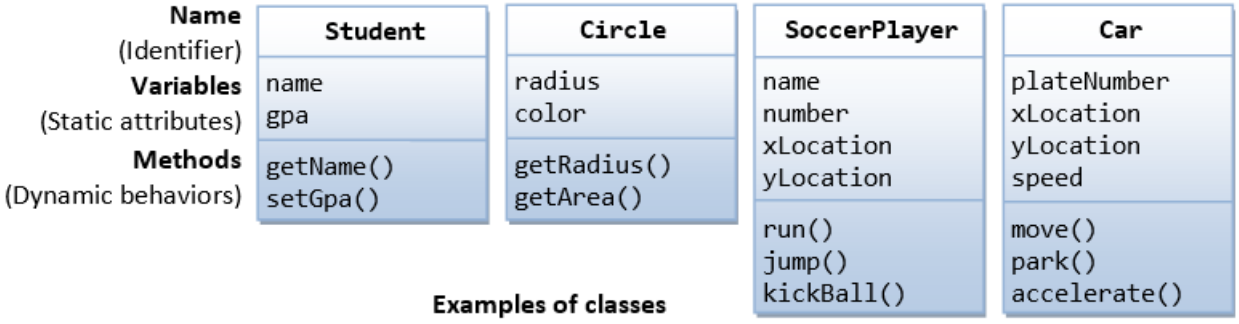
A class is a 3-compartment box

A class can be visualized as a three-compartment box, as illustrated:

1. *Name* (or identity): identifies the class.
2. *Variables* (or attribute, state, field): contains the *static attributes* of the class.
3. *Methods* (or behaviors, function, operation): contains the *dynamic behaviors* of the class.

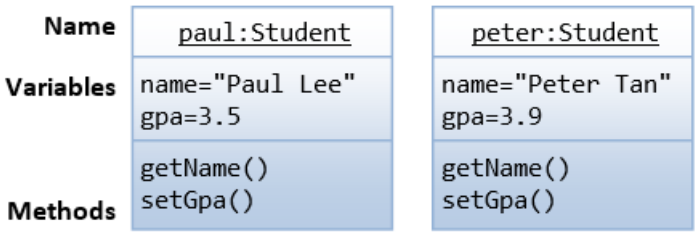
In other words, a class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.

The followings figure shows a few examples of classes:



Examples of classes

The following figure shows two instances of the class `Student`, identified as "`paul`" and "`peter`".



Two instances - paul and peter - of the class Student

Unified Modeling Language (UML) Class and Instance Diagrams: The above class diagrams are drawn according to the UML notations. A class is represented as a 3-compartment box, containing name, variables, and methods, respectively. Class name is

shown in bold and centralized. An instance is also represented as a 3-compartment box, with instance name shown as *instanceName:Classname* and underlined.

Brief Summary

1. A *class* is a programmer-defined, abstract, self-contained, reusable software entity that mimics a real-world thing.
2. A class is a 3-compartment box containing the name, variables and the methods.
3. A class encapsulates the data structures (in variables) and algorithms (in methods). The values of the variables constitute its *state*. The methods constitute its *behaviors*.
4. An *instance* is an instantiation (or realization) of a particular item of a class.

2.3 Class Definition in Java

In Java, we use the keyword `class` to define a class. For examples:

```
public class Circle {        // class name
   double radius;            // variables
   String color;

   double getRadius() { ..... }  // methods
   double getArea() { ..... }
}
```

```
public class SoccerPlayer {  // class name
   int number;               // variables
   String name;
   int x, y;

   void run() { ..... }      // methods
   void kickBall() { ..... }
}
```

The syntax for class definition in Java is:

```
[AccessControlModifier] class ClassName {
   // Class body contains members (variables and methods)
   ......
}
```

We shall explain the *access control modifier*, such as `public` and `private`, later.

Class Naming Convention: A class name shall be a noun or a noun phrase made up of several words. All the words shall be initial-capitalized (camel-case). Use a *singular* noun for class name. Choose a meaningful and self-descriptive classname. For examples, `SoccerPlayer`, `HttpProxyServer`, `FileInputStream`, `PrintStream` and `SocketFactory`.

2.4 Creating Instances of a Class

To create *an instance of a class*, you have to:

1. **Declare** an instance identifier (instance name) of a particular class.
2. **Construct** the instance (i.e., allocate storage for the instance and initialize the instance) using the "`new`" operator.

For examples, suppose that we have a class called `Circle`, we can create instances of `Circle` as follows:

```
// Declare 3 instances of the class Circle, c1, c2, and c3
Circle c1, c2, c3;  // They hold a special value called null
// Construct the instances via new operator
c1 = new Circle();
c2 = new Circle(2.0);
c3 = new Circle(3.0, "red");

// You can Declare and Construct in the same statement
Circle c4 = new Circle();
```

When an instance is declared but not constructed, it holds a special value called `null`.

2.5 Dot (.) Operator

The *variables* and *methods* belonging to a class are formally called *member variables* and *member methods*. To reference a member variable or method, you must:

1. First identify the instance you are interested in, and then,
2. Use the *dot operator* (`.`) to reference the desired member variable or method.

For example, suppose that we have a class called `Circle`, with two member variables (`radius` and `color`) and two member methods (`getRadius()` and `getArea()`). We have created three instances of the class `Circle`, namely, `c1`, `c2` and `c3`. To invoke the method `getArea()`, you must first identity the instance of interest, says `c2`, then use the *dot operator*, in the form of `c2.getArea()`.

For example,

```
// Suppose that the class Circle has variables radius and color,
//  and methods getArea() and getRadius().
// Declare and construct instances c1 and c2 of the class Circle
Circle c1 = new Circle ();
Circle c2 = new Circle ();
// Invoke member methods for the instance c1 via dot operator
System.out.println(c1.getArea());
System.out.println(c1.getRadius());
// Reference member variables for instance c2 via dot operator
c2.radius = 5.0;
c2.color = "blue";
```

Calling `getArea()` without identifying the instance is meaningless, as the radius is unknown (there could be many instances of `Circle` - each maintaining its own radius). Furthermore, `c1.getArea()` and `c2.getArea()` are likely to produce different results.

In general, suppose there is a class called *`AClass`* with a member variable called *`aVariable`* and a member method called *`aMethod`*`()`. An instance called *`anInstance`* is constructed for *`AClass`*. You use *`anInstance.aVariable`* and *`anInstance.aMethod`*`()`.

2.6 Member Variables

A *member variable* has a *name* (or *identifier*) and a *type*; and holds a *value* of that particular type (as descried in the earlier chapter).

Variable Naming Convention: A variable name shall be a noun or a noun phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case), e.g., `fontSize`, `roomNumber`, `xMax`, `yMin` and `xTopLeft`.

The formal syntax for variable definition in Java is:

```
[AccessControlModifier] type variableName [= initialValue];
[AccessControlModifier] type variableName-1 [= initialValue-1] [, type variableName-2 [= initialValue-2]] ... ;
```

For example,

```
private double radius;
public int length = 1, width = 1;
```

2.7 Member Methods

A method (as described in the earlier chapter):

1. receives arguments from the caller,
2. performs the operations defined in the method body, and
3. returns a piece of result (or `void`) to the caller.

The syntax for method declaration in Java is as follows:

```
[AccessControlModifier] returnType methodName ([parameterList]) {
   // method body or implementation
   ......
}
```

For examples:

```
// Return the area of this Circle instance
public double getArea() {
   return radius * radius * Math.PI;
}
```

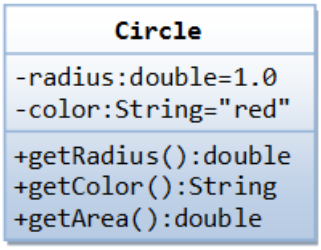
Method Naming Convention: A method name shall be a verb, or a verb phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case). For example, `getArea()`, `setRadius()`, `getParameterValues()`, `hasNext()`.

Variable name vs. Method name vs. Class name: A variable name is a noun, denoting an attribute; while a method name is a verb, denoting an action. They have the same naming convention (the first word in lowercase and the rest are initial-capitalized). Nevertheless, you can easily distinguish them from the context. Methods take arguments in parentheses (possibly zero arguments with empty parentheses), but variables do not. In this writing, methods are denoted with a pair of parentheses, e.g., `println()`, `getArea()` for clarity.

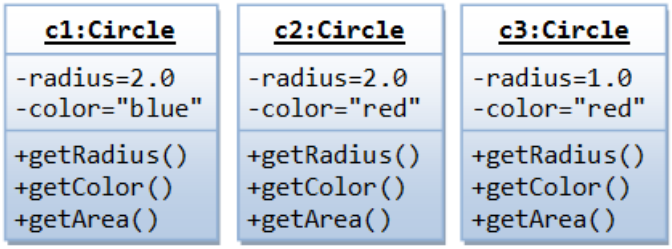
On the other hand, class name is a noun beginning with uppercase.

2.8 Putting them Together: An OOP Example

Class Definition



Instances



A class called `Circle` is defined as shown in the class diagram. It contains two `private` member variables: `radius` (of type `double`) and `color` (of type `String`); and three `public` member methods: `getRadius()`, `getColor()`, and `getArea()`.

Three instances of `Circle`s, called `c1`, `c2`, and `c3`, shall be constructed with their respective data members, as shown in the instance diagrams.

The source codes for `Circle.java` is as follows:

`Circle.java`

```
1  /*
2   * The Circle class models a circle with a radius and color.
3   */
4  public class Circle {    // Save as "Circle.java"
5     // Private instance variables
6     private double radius;
7     private String color;
8
9     // Constructors (overloaded)
10    public Circle() {                  // 1st Constructor
11       radius = 1.0;
12       color = "red";
13    }
14    public Circle(double r) {          // 2nd Constructor
15       radius = r;
16       color = "red";
17    }
```

```
18      public Circle(double r, String c) { // 3rd Constructor
19         radius = r;
20         color = c;
21      }
22
23      // Public methods
24      public double getRadius() {
25         return radius;
26      }
27      public String getColor() {
28         return color;
29      }
30      public double getArea() {
31         return radius * radius * Math.PI;
32      }
33   }
```

Compile "`Circle.java`" into "`Circle.class`".

Notice that the `Circle` class does not have a `main()` method. Hence, it is NOT a standalone program and you cannot run the `Circle` class by itself. The `Circle` class is meant to be a building block - to be used in other programs.

`TestCircle.java`

We shall now write another class called `TestCircle`, which uses the `Circle` class. The `TestCircle` class has a `main()` method and can be executed.

```
1   /*
2    * A Test Driver for the "Circle" class
3    */
4   public class TestCircle {    // Save as "TestCircle.java"
5      public static void main(String[] args) {   // Program entry point
6         // Declare and Construct an instance of the Circle class called c1
7         Circle c1 = new Circle(2.0, "blue");  // Use 3rd constructor
8         System.out.println("The radius is: " + c1.getRadius());  // use dot operator to invoke member methods
9         System.out.println("The color is: " + c1.getColor());
10        System.out.printf("The area is: %.2f%n", c1.getArea());
11
12        // Declare and Construct another instance of the Circle class called c2
13        Circle c2 = new Circle(2.0);  // Use 2nd constructor
14        System.out.println("The radius is: " + c2.getRadius());
15        System.out.println("The color is: " + c2.getColor());
16        System.out.printf("The area is: %.2f%n", c2.getArea());
17
18        // Declare and Construct yet another instance of the Circle class called c3
19        Circle c3 = new Circle();  // Use 1st constructor
20        System.out.println("The radius is: " + c3.getRadius());
21        System.out.println("The color is: " + c3.getColor());
22        System.out.printf("The area is: %.2f%n", c3.getArea());
23     }
24  }
```

Compile `TestCircle.java` into `TestCircle.class`.

Run the `TestCircle` and study the output:

```
The radius is: 2.0
The color is: blue
The area is: 12.57
The radius is: 2.0
The color is: red
The area is: 12.57
The radius is: 1.0
The color is: red
The area is: 3.14
```

2.9 Constructors

A *constructor* looks like a special method that has the *same method name as the class name*. In the above `Circle` class, we define three overloaded versions of constructor `Circle(.....)`. A constructor is used to *construct* and *initialize* all the member variables. To construct a new instance of a class, you need to use a special "`new`" operator followed by a call to one of the constructors. For example,

```
Circle c1 = new Circle();
Circle c2 = new Circle(2.0);
Circle c3 = new Circle(3.0, "red");
```

A constructor is different from an ordinary method in the following aspects:

- The name of the constructor method is the same as the class name. By classname's convention, it begins with an uppercase (instead of lowercase for ordinary methods).
- Constructor has no return type. It implicitly returns `void`. No `return` statement is allowed inside the constructor's body.
- Constructor can only be invoked via the "`new`" operator. It can only be used *once* to initialize the instance constructed. Once an instance is constructed, you cannot call the constructor anymore.
- Constructors are not inherited (to be explained later).

Default Constructor: A constructor with no parameter is called the *default constructor*. It initializes the member variables to their default value. For example, the `Circle()` in the above example initialize member variables `radius` and `color` to their default value.

2.10 Method Overloading (Revisit)

Method overloading means that the *same method name* can have *different implementations* (versions). However, the different implementations must be distinguishable by their parameter list (either the number of parameters, or the type of parameters, or their order).

Example: The method `average()` has 3 versions, with different parameter lists. The caller can invoke the chosen version by supplying the matching arguments.

```
1  /*
2   * Example to illustrate Method Overloading
3   */
4  public class TestMethodOverloading {
5     public static int average(int n1, int n2) {          // version A
6        System.out.println("Run version A");
7        return (n1+n2)/2;
8     }
9
10    public static double average(double n1, double n2) { // version B
11       System.out.println("Run version B");
12       return (n1+n2)/2;
13    }
14
15    public static int average(int n1, int n2, int n3) {  // version C
16       System.out.println("Run version C");
17       return (n1+n2+n3)/3;
18    }
19
20    public static void main(String[] args) {
21       System.out.println(average(1, 2));     // Use A
22       System.out.println(average(1.0, 2.0)); // Use B
23       System.out.println(average(1, 2, 3));  // Use C
24       System.out.println(average(1.0, 2));   // Use B - int 2 implicitly casted to double 2.0
25       // average(1, 2, 3, 4); // Compilation Error - No matching method
26    }
27 }
```

Overloading `Circle` Class' Constructor

Constructor, like an ordinary method, can also be overloaded. The above `Circle` class has three overloaded versions of constructors differentiated by their parameter list, as followed:

```
Circle()
Circle(double r)
Circle(double r, String c)
```

Depending on the actual argument list used when invoking the method, the matching constructor will be invoked. If your argument list does not match any one of the methods, you will get a compilation error.

2.11 `public` vs. `private` - Access Control Modifiers

An *access control modifier* can be used to *control the visibility* of a class, or a member variable or a member method within a class. We begin with the following two access control modifiers:

1. `public`: The class/variable/method is accessible and available to *all* the other objects in the system.
2. `private`: The class/variable/method is accessible and available *within this class only*.

For example, in the above `Circle` definition, the member variable `radius` is declared `private`. As the result, `radius` is accessible inside the `Circle` class, but NOT in the `TestCircle` class. In other words, you cannot use "`c1.radius`" to refer to `c1`'s `radius` in `TestCircle`.

- Try inserting the statement "`System.out.println(c1.radius)`" in `TestCircle` and observe the error message.
- Try changing `radius` to `public` in the `Circle` class, and re-run the above statement.

On the other hand, the method `getRadius()` is declared `public` in the `Circle` class. Hence, it can be invoked in the `TestCircle` class.

UML Notation: In UML class diagram, `public` members are denoted with a "`+`"; while `private` members with a "`-`".

More access control modifiers will be discussed later.

2.12 Information Hiding and Encapsulation

A class encapsulates the name, static attributes and dynamic behaviors into a "3-compartment box". Once a class is defined, you can seal up the "box" and put the "box" on the shelve for others to use and reuse. Anyone can pick up the "box" and use it in their application. This cannot be done in the traditional procedural-oriented language like C, as the static attributes (or variables) are scattered over the entire program and header files. You cannot "cut" out a portion of C program, plug into another program and expect the program to run without extensive changes.

Member variables of a class are typically hidden from the outside word (i.e., the other classes), with `private` access control modifier. Access to the member variables are provided via `public` assessor methods, e.g., `getRadius()` and `getColor()`.

This follows the principle of *information hiding*. That is, objects communicate with each others using well-defined interfaces (public methods). Objects are not allowed to know the implementation details of others. The implementation details are hidden or encapsulated within the class. Information hiding facilitates reuse of the class.

Rule of Thumb: Do not make any variable `public`, unless you have a good reason.

2.13 The `public` Getters and Setters for `private` Variables

To allow other classes to *read* the value of a `private` variable says `xxx`, we provide a *get method* (or *getter* or *accessor method*) called `getXxx()`. A get method needs not expose the data in raw format. It can process the data and limit the view of the data others will see. The getters shall not modify the variable.

To allow other classes to *modify* the value of a `private` variable says `xxx`, we provide a *set method* (or *setter* or *mutator method*) called `setXxx()`. A set method could provide data validation (such as range checking), or transform the raw data into the internal representation.

For example, in our `Circle` class, the variables `radius` and `color` are declared `private`. That is to say, they are only accessible within the `Circle` class and not visible in any other classes, such as the `TestCircle` class. You cannot access the `private` variables `radius` and `color` from the `TestCircle` class directly - via says `c1.radius` or `c1.color`. The `Circle` class provides two public accessor methods, namely, `getRadius()` and `getColor()`. These methods are declared `public`. The class `TestCircle` can invoke these public accessor methods to retrieve the `radius` and `color` of a `Circle` object, via says `c1.getRadius()` and `c1.getColor()`.

There is no way you can change the `radius` or `color` of a `Circle` object, after it is constructed in the `TestCircle` class. You cannot issue statements such as `c1.radius = 5.0` to change the `radius` of instance `c1`, as `radius` is declared as `private` in the `Circle` class and is not visible to other classes including `TestCircle`.

If the designer of the `Circle` class permits the change the `radius` and `color` after a `Circle` object is constructed, he has to provide the appropriate set methods (or setters or mutator methods), e.g.,

```
// Setter for color
public void setColor(String newColor) {
```

```
   color = newColor;
}

// Setter for radius
public void setRadius(double newRadius) {
   radius = newRadius;
}
```

With proper implementation of *information hiding*, the designer of a class has full control of what the user of the class can and cannot do.

2.14 Keyword "this"

You can use keyword "`this`" to refer to *this* instance inside a class definition.

One of the main usage of keyword `this` is to resolve ambiguity.

```
public class Circle {
   double radius;                // Member variable called "radius"
   public Circle(double radius) { // Method's argument also called "radius"
      this.radius = radius;
         // "radius = radius" does not make sense!
         // "this.radius" refers to this instance's member variable
         // "radius" resolved to the method's argument.
   }
   ...
}
```

In the above codes, there are two identifiers called `radius` - a member variable of the class and the method's argument. This causes naming conflict. To avoid the naming conflict, you could name the method's argument `r` instead of `radius`. However, `radius` is more approximate and meaningful in this context. Java provides a keyword called `this` to resolve this naming conflict. "`this.radius`" refers to the member variable; while "`radius`" resolves to the method's argument.

Using the keyword "`this`", the constructor, getter and setter methods for a `private` variable called `xxx` of type `T` are as follows:

```
public class Aaa {
   // A private variable named xxx of the type T
   private T xxx;

   // Constructor
   public Aaa(T xxx) {
      this.xxx = xxx;
   }

   // A getter for variable xxx of type T receives no argument and return a value of type T
   public T getXxx() {
      return xxx;  // or "return this.xxx" for clarity
   }

   // A setter for variable xxx of type T receives a parameter of type T and return void
   public void setXxx(T xxx) {
      this.xxx = xxx;
   }
}
```

For a boolean variable `xxx`, the getter shall be named `isXxx()` or `hasXxx()`, which is more meaningful than `getXxx()`. The setter remains `setXxx()`.

```
// Private boolean variable
private boolean xxx;

// Getter
public boolean isXxx() {
   return xxx;  // or "return this.xxx" for clarity
}

// Setter
public void setXxx(boolean xxx) {
   this.xxx = xxx;
}
```

More on "this"

- `this.varName` refers to *varName* of this instance; `this.methodName(...)` invokes *methodName*`(...)` of this instance.
- In a constructor, we can use `this(...)` to call *another* constructor of this class.
- Inside a method, we can use the statement "`return this`" to return this instance to the caller.

2.15 Method `toString()`

Every well-designed Java class should have a `public` method called `toString()` that returns a `string` description of `this` instance. You can invoke the `toString()` method explicitly by calling *anInstanceName*`.toString()`, or implicitly via `println()` or `String` concatenation operator '`+`'. That is, running `println(`*anInstance*`)` invokes the `toString()` method of that instance implicitly.

For example, include the following `toString()` method in our `Circle` class:

```
// Return a String description of this instance
public String toString() {
   return "Circle[radius=" + radius + ",color=" + color + "]";
}
```

In your `TestCircle` class, you can get a description of a `Circle` instance via:

```
Circle c1 = new Circle();
System.out.println(c1.toString());   // Explicitly calling toString()
System.out.println(c1);              // Implicit call to c1.toString()
System.out.println("c1 is: " + c1);  // '+' invokes toString() to get a String before concatenation
```

The signature of `toString()` is:

```
public String toString() { ..... }
```

2.16 Constants (`final`)

Constants are variables defined with the modifier `final`. A `final` variable can only be assigned once and its value cannot be modified once assigned. For example,

```
public final double X_REFERENCE = 1.234;

private final int MAX_ID = 9999;
MAX_ID = 10000;  // error: cannot assign a value to final variable MAX_ID

// You need to initialize a final member variable during declaration
private final int SIZE;    // error: variable SIZE might not have been initialized
```

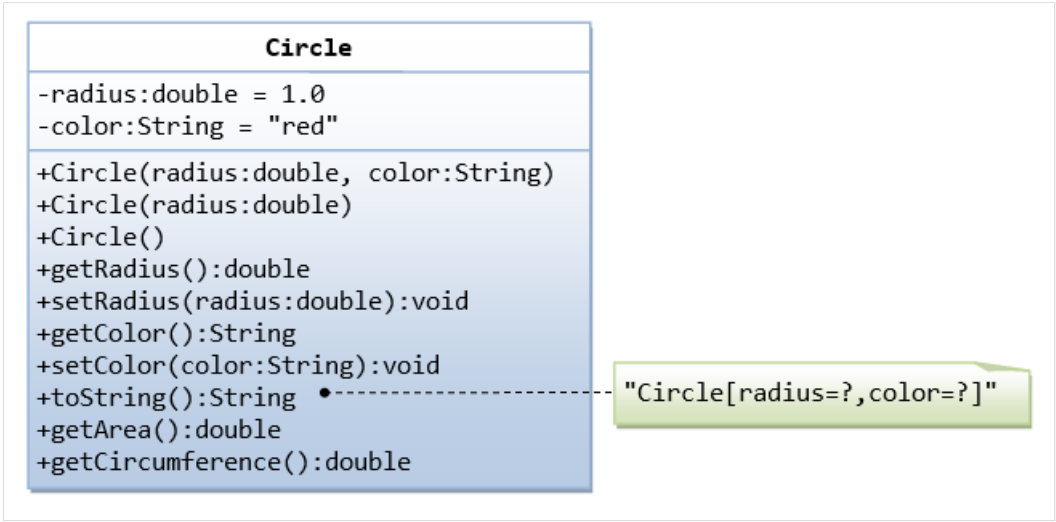
Constant Naming Convention: A constant name is a noun, or a noun phrase made up of several words. All words are in uppercase separated by underscores '_', for examples, `X_REFERENCE`, `MAX_INTEGER` and `MIN_VALUE`.

Advanced Notes:

1. A `final` primitive variable cannot be re-assigned a new value.
2. A `final` instance cannot be re-assigned a new object.
3. A `final` class cannot be sub-classed (or extended).
4. A `final` method cannot be overridden.

2.17 Putting Them Together in the Revised `Circle` Class

We shall include constructors, getters, setters, `toString()`, and use the keyword "`this`". The class diagram for the final `Circle` class is as follows:



Circle.java

```java
/*
 * The Circle class models a circle with a radius and color.
 */
public class Circle {    // Save as "Circle.java"
   // The public constants
   public static final double DEFAULT_RADIUS = 8.8;
   public static final String DEFAULT_COLOR  = "red";

   // The private instance variables
   private double radius;
   private String color;

   // The (overloaded) constructors
   public Circle() {                   // 1st (default) Constructor
      this.radius = DEFAULT_RADIUS;
      this.color  = DEFAULT_COLOR;
   }
   public Circle(double radius) {      // 2nd Constructor
      this.radius = radius;
      this.color = DEFAULT_COLOR;
   }
   public Circle(double radius, String color) { // 3rd Constructor
      this.radius = radius;
      this.color = color;
   }

   // The public getters and setters for the private variables
   public double getRadius() {
      return this.radius;
   }
   public void setRadius(double radius) {
      this.radius = radius;
   }
   public String getColor() {
      return this.color;
   }
   public void setColor(String color) {
      this.color = color;
   }

   // The toString() returns a String description of this instance
   public String toString() {
      return "Circle[radius=" + radius + ", color=" + color + "]";
   }

   // Return the area of this Circle
```

```
47      public double getArea() {
48         return radius * radius * Math.PI;
49      }
50
51      // Return the circumference of this Circle
52      public double getCircumference() {
53         return 2.0 * radius * Math.PI;
54      }
55   }
```

A Test Driver for the `Circle` Class

```
// A Test Driver for the Circle class
public class TestCircle {
   public static void main(String[] args) {
      // Test constructors and toString()
      Circle c1 = new Circle(1.1, "blue");
      System.out.println(c1);  // toString()
      Circle c2 = new Circle(2.2);
      System.out.println(c2);  // toString()
      Circle c3 = new Circle();
      System.out.println(c3);  // toString()

      // Test Setters and Getters
      c1.setRadius(2.2);
      c1.setColor("green");
      System.out.println(c1);  // toString() to inspect the modified instance
      System.out.println("The radius is: " + c1.getRadius());
      System.out.println("The color is: " + c1.getColor());

      // Test getArea() and getCircumference()
      System.out.printf("The area is: %.2f%n", c1.getArea());
      System.out.printf("The circumference is: %.2f%n", c1.getCircumference());
   }
}
```

The expected outputs are:

```
Circle[radius=1.1, color=blue]
Circle[radius=2.2, color=red]
Circle[radius=8.8, color=red]
Circle[radius=2.2, color=green]
Radius is: 2.2
Color is: green
Area is: 15.21
Circumference is: 13.82
```