Java Programming Tutorial

Object-oriented Programming (OOP) Basics

TABLE OF CONTENTS (HIDE)

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*.

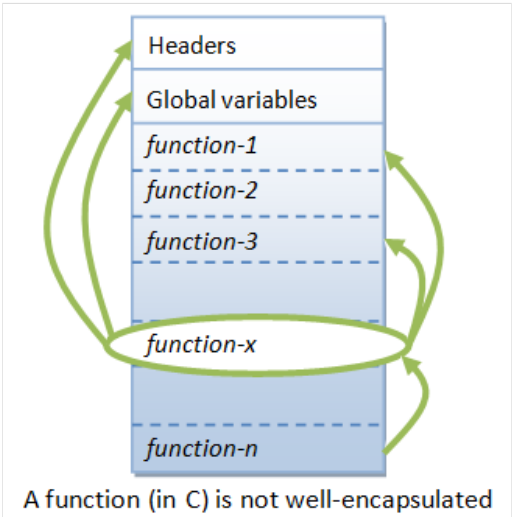
Similarly, a car is assembled from parts and components, such as chassis, doors, engine, wheels, brake and transmission. The components are reusable, e.g., a wheel can be used in many cars (of the same specifications).

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).

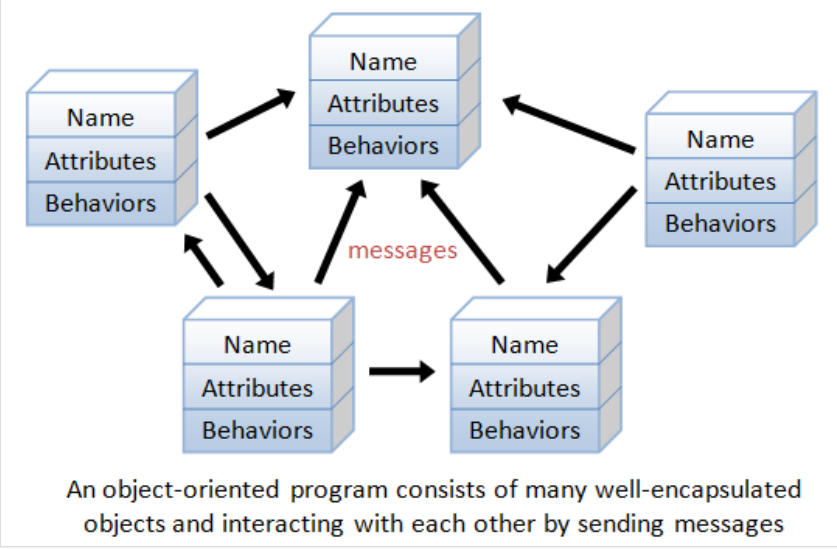
In the early 1970s, the US Department of Defense (DoD) commissioned a task force to investigate why its IT budget always went out of control; but without much to show for. The findings are:

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

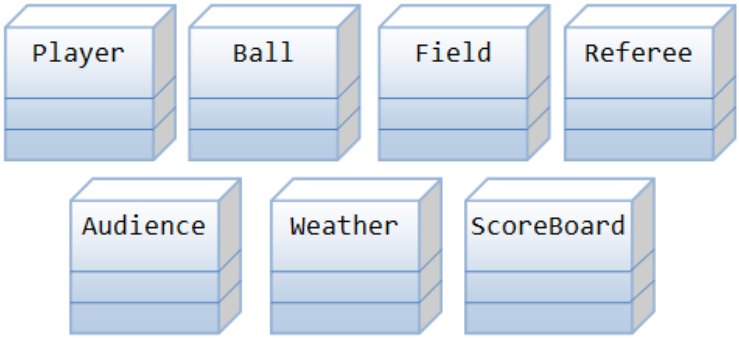
Object-oriented programming (OOP) languages are designed to overcome these problems.



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, x and y location in the field, and etc; operations include run, jump, kick-the-ball, and etc.
- Ball: attributes include x, y, z position in the field, radius, weight, etc.
- Referee:
- 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

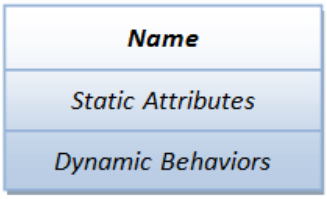
2.1 Class & Instances

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 "`Alice`", "`Ah Beng`" and "`Ali`".

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



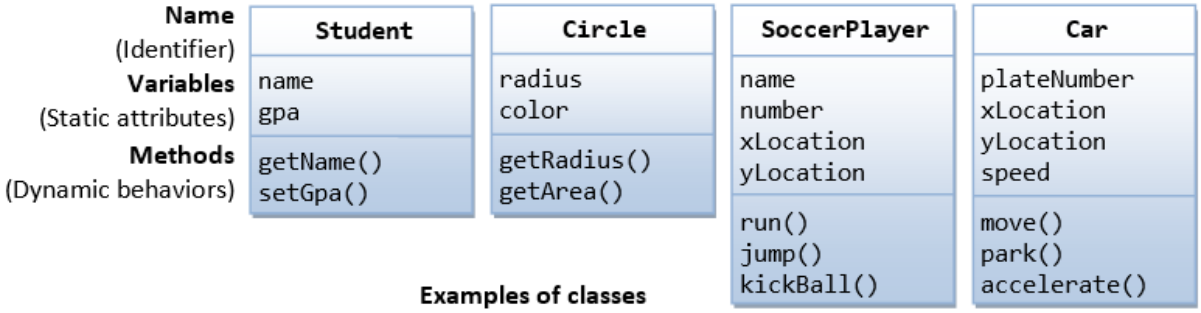
A class is a 3-compartment box

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.

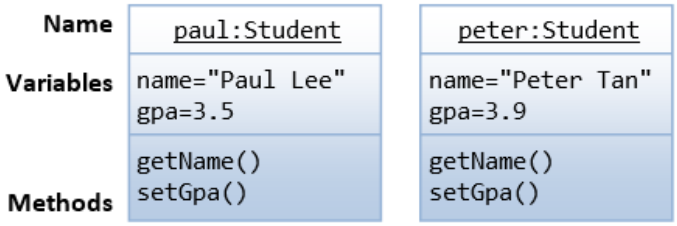
In other words, a class encapsulates the static attributes (data) and dynamic behaviors (operations that operate on the data) in a box.

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, say `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;
}
```

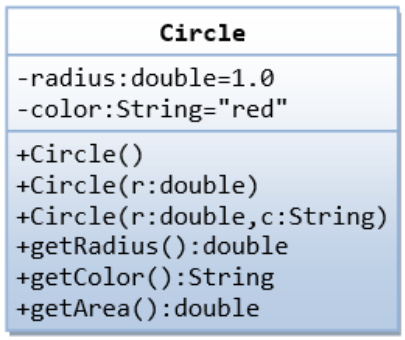
Method Naming Convention: A method name shall be a verb, or a verb phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case). For example, `getArea()`, `setRadius()`, `getParameterValues()`, `hasNext()`.

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.

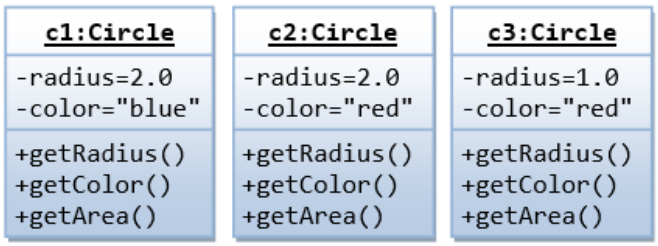
On the other hand, class name is a noun beginning with uppercase.

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`

```
/**
 * The Circle class models a circle with a radius and color.
 */
public class Circle {    // Save as "Circle.java"
   // Private instance variables
   private double radius;
   private String color;

   // Constructors (overloaded)
   /** Constructs a Circle instance with default radius and color */
   public Circle() {                  // 1st Constructor (default constructor)
      radius = 1.0;
      color = "red";
   }
   /** Constructs a Circle instance with the given radius and default color*/
   public Circle(double r) {          // 2nd Constructor
      radius = r;
      color = "red";
   }
   /** Constructs a Circle instance with the given radius and color */
   public Circle(double r, String c) { // 3rd Constructor
      radius = r;
      color = c;
   }

   // Public methods
   /** Returns the radius */
   public double getRadius() {  // getter for radius
      return radius;
   }
   /** Returns the color */
   public String getColor() {   // getter for color
      return color;
   }
   /** Returns the area of this circle */
   public double getArea() {
      return radius * radius * Math.PI;
```

```
38      }
39  }
```

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

```
cd \path\to\project-directory
javac Circle.java
```

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        //The radius is: 2.0
10       System.out.println("The color is: " + c1.getColor());
11       //The color is: blue
12       System.out.printf("The area is: %.2f%n", c1.getArea());
13       //The area is: 12.57
14
15       // Declare and Construct another instance of the Circle class called c2
16       Circle c2 = new Circle(2.0);  // Use 2nd constructor
17       System.out.println("The radius is: " + c2.getRadius());
18       //The radius is: 2.0
19       System.out.println("The color is: " + c2.getColor());
20       //The color is: red
21       System.out.printf("The area is: %.2f%n", c2.getArea());
22       //The area is: 12.57
23
24       // Declare and Construct yet another instance of the Circle class called c3
25       Circle c3 = new Circle();  // Use 1st constructor
26       System.out.println("The radius is: " + c3.getRadius());
27       //The radius is: 1.0
28       System.out.println("The color is: " + c3.getColor());
29       //The color is: red
30       System.out.printf("The area is: %.2f%n", c3.getArea());
31       //The area is: 3.14
32    }
33 }
```

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

```
javac TestCircle.java
```

Run the `TestCircle` and study the outputs (shown in blue).

```
java TestCircle
```

2.9 Constructors

A *constructor* is a special method that has the *same method name as the class name*. That is, the constructor of the class `Circle` is called `Circle()`. 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();           // use 1st constructor
Circle c2 = new Circle(2.0);        // use 2nd constructor
Circle c3 = new Circle(3.0, "red"); // use 3rd constructor
```

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

- The *name* of the constructor method must be the same as the classname. By classname's convention, it begins with an uppercase (instead of lowercase for ordinary methods).
- Constructor has *no return type* in its method heading. 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). Every class shall define its own constructors.

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

2.10 Revisit Method Overloading

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 MethodOverloadingTest {
5     public static int average(int n1, int n2) {          // version 1
6        System.out.println("Run version 1");
7        return (n1+n2)/2;
8     }
9     public static double average(double n1, double n2) { // version 2
10       System.out.println("Run version 2");
11       return (n1+n2)/2;
12    }
13    public static int average(int n1, int n2, int n3) {  // version 3
14       System.out.println("Run version 3");
15       return (n1+n2+n3)/3;
16    }
17
18    public static void main(String[] args) {
19       System.out.println(average(1, 2));
20       //Run version 1
21       //1
22       System.out.println(average(1.0, 2.0));
23       //Run version 2
24       //1.5
25       System.out.println(average(1, 2, 3));
26       //Run version 3
27       //2
28       System.out.println(average(1.0, 2));
29       //Run version 2 (int 2 implicitly casted to double 2.0)
30       //1.5
31
32       //average(1, 2, 3, 4);
33       //compilation error: no suitable method found for average(int,int,int,int)
34    }
35 }
```

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()  // the default constructor
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.

Note: C language does not support method overloading. You need to use different method names for each of the variations. C++, Java, C# support method overloading.

2.11 The Access Control Modifiers: `public`/`private`

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 (error: `radius` has `private` access in `Circle`).
- 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, e.g., `c1.getRadius()`.

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 variables `public`, unless you have a good reason.

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

To allow other classes to *read* the value of a `private` variable say `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 say `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, including the `TestCircle` class. You cannot access the `private` variables `radius` and `color` from the `TestCircle` class directly - via say `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 say `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 parameter 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 parameter.
   }
   ...
}
```

In the above codes, there are two identifiers called `radius` - a member variable of the class and the method's parameter. 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 Ccc {
   // A private variable named xxx of the type T
   private T xxx;

   // Constructor
   public Ccc(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 as `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 shall 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 c4 = new Circle();
System.out.println(c4.toString());   // Explicitly calling toString()
//Circle[radius=1.0,color=red]
System.out.println(c4);              // Implicit call to c4.toString()
//Circle[radius=1.0,color=red]
System.out.println("c4 is: " + c4);  // '+' invokes toString() to get a String before concatenation
//Circle[radius=1.0,color=red]
```

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;  //compilation error: cannot assign a value to final variable MAX_ID

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

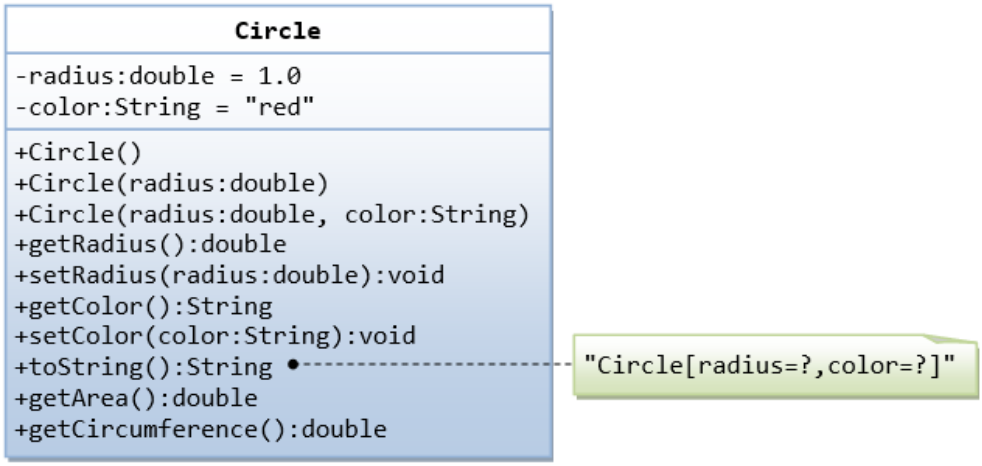
Constant Naming Convention: A constant name is a noun, or a noun phrase made up of several words. All words are in uppercase separated by underscores '_', for examples, X_REFERENCE, MAX_INTEGER and MIN_VALUE.

Advanced notes for final:

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 Finalized 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

```
1  /**
```

```
 * 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 = 1.0;
   public static final String DEFAULT_COLOR  = "red";

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

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

   /** Returns the radius - the public getter for private variable radius. */
   public double getRadius() {
      return this.radius;
   }
   /** Sets the radius - the public setter for private variable radius */
   public void setRadius(double radius) {
      this.radius = radius;
   }
   /** Returns the color - the public getter for private variable color */
   public String getColor() {
      return this.color;
   }
   /** Sets the color - the public setter for private variable color */
   public void setColor(String color) {
      this.color = color;
   }

   /** Returns a self-descriptive string for this Circle instance */
   public String toString() {
      return "Circle[radius=" + radius + ", color=" + color + "]";
   }

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

   /** Returns the circumference of this Circle */
   public double getCircumference() {
      return 2.0 * radius * Math.PI;
   }
}
```

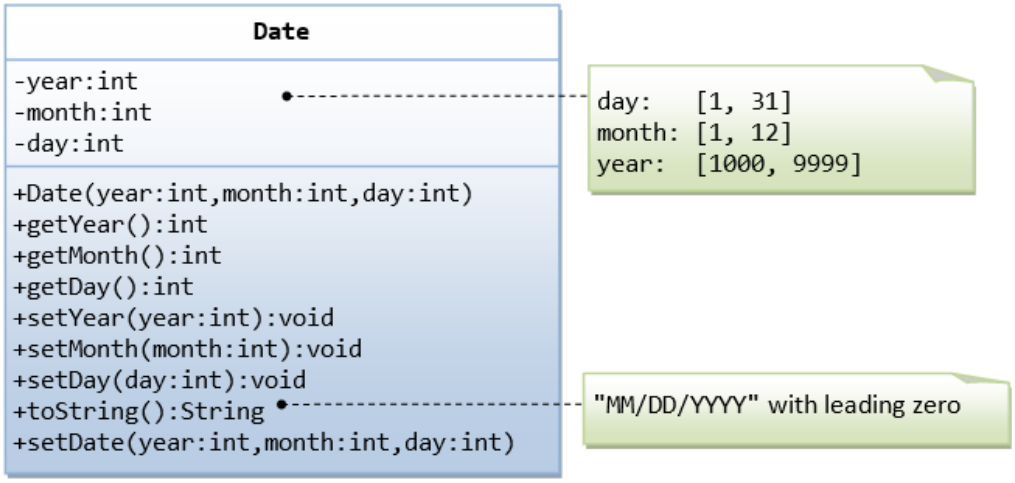
A Test Driver for the `Circle` Class (`TestCircle.java`)

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

```
      Circle c2 = new Circle(2.2);
      System.out.println(c2);
      //Circle[radius=2.2, color=red]
      Circle c3 = new Circle();
      System.out.println(c3);
      //Circle[radius=1.0, color=red]

      // Test Setters and Getters
      c1.setRadius(3.3);
      c1.setColor("green");
      System.out.println(c1);  // use toString() to inspect the modified instance
      //Circle[radius=3.3, color=green]
      System.out.println("The radius is: " + c1.getRadius());
      //The radius is: 3.3
      System.out.println("The color is: " + c1.getColor());
      //The color is: green

      // Test getArea() and getCircumference()
      System.out.printf("The area is: %.2f%n", c1.getArea());
      //The area is: 34.21
      System.out.printf("The circumference is: %.2f%n", c1.getCircumference());
      //The circumference is: 20.73
   }
}
```

3. More Examples on Classes

3.1 EG. 1: The `Date` class



A `Date` class models a calendar date with day, month and year, is designed as shown in the class diagram. It contains the following members:

- 3 `private` instance variables `day`, `month`, and `year`.
- Constructors, `public` getters and setters for the `private` instance variables.
- A method `setDate()`, which sets the `day`, `month` and `year`.
- A `toString()`, which returns "`DD/MM/YYYY`", with leading zero for `DD` and `MM` if applicable.

Write the `Date` class and a test driver to test all the `public` methods. No Input validations are required for `day`, `month`, and `year`.

The `Date` Class (`Date.java`)

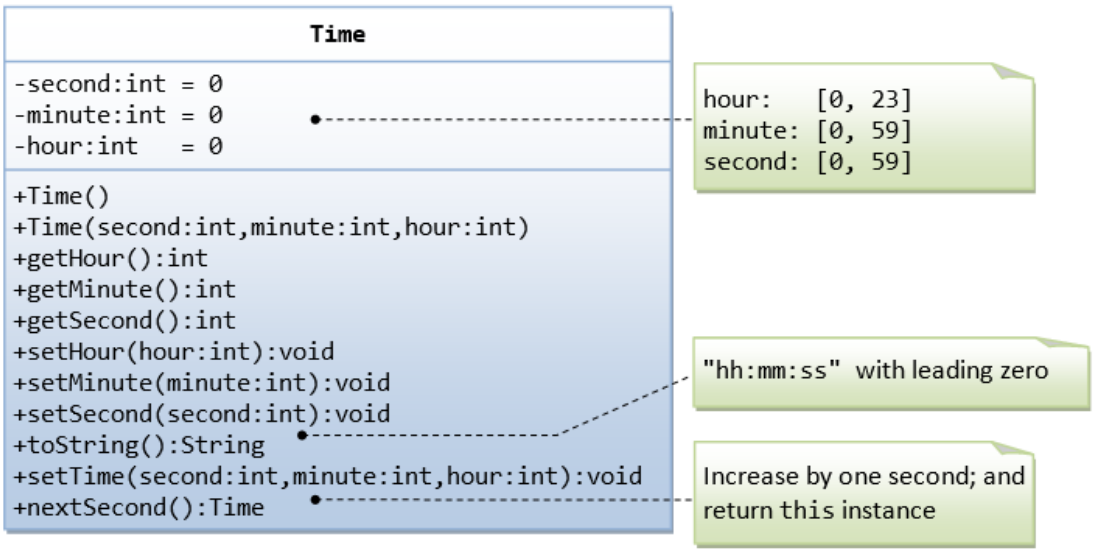
```
/**
 * The Date class models a calendar date with day, month and year.
 * This class does not perform input validation for day, month and year.
 */
public class Date {
   // The private instance variables
   private int year, month, day;

   /** Constructs a Date instance with the given year, month and day. No input validation */
   public Date(int year, int month, int day) {
      this.year = year;
```

```
      this.month = month;
      this.day = day;
   }

   // The public getters/setters for the private variables
   /** Returns the year */
   public int getYear() {
      return this.year;
   }
   /** Returns the month */
   public int getMonth() {
      return this.month;
   }
   /** Returns the day */
   public int getDay() {
      return this.day;
   }
   /** Sets the year. No input validation */
   public void setYear(int year) {
      this.year = year;
   }
   /** Sets the month. No input validation */
   public void setMonth(int month) {
      this.month = month;
   }
   /** Sets the day. No input validation */
   public void setDay(int day) {
      this.day = day;
   }

   /* Returns a descriptive String in the form "MM/DD/YYYY" with leading zero */
   public String toString() {
        // Use built-in function String.format() to form a formatted String
        return String.format("%02d/%02d/%4d", month, day, year);
              // Specifier "0" to print leading zeros
   }

   /** Sets year, month and day. No input validation */
   public void setDate(int year, int month, int day) {
      this.year = year;
      this.month = month;
      this.day = day;
   }
}
```

A Test Driver for the `Date` Class (`TestDate.java`)

```
/**
 * A Test Driver for the Date class.
 */
public class TestDate {
   public static void main(String[] args) {
      // Test constructor and toString()
      Date d1 = new Date(2020, 2, 8);
      System.out.println(d1);  // toString()
      //02/08/2020

      // Test Setters and Getters
      d1.setYear(2012);
      d1.setMonth(12);
      d1.setDay(23);
      System.out.println(d1);
      //12/23/2012
      System.out.println("Year is: " + d1.getYear());
      //Year is: 2012
      System.out.println("Month is: " + d1.getMonth());
      //Month is: 12
      System.out.println("Day is: " + d1.getDay());
      //Day is: 23

      // Test setDate()
      d1.setDate(2988, 1, 2);
      System.out.println(d1);
      //01/02/2988
   }
}
```

3.2 EG. 2: The `Time` class



A class called `Time`, which models a time instance with hour, minute and second, is designed as shown in the class diagram. It contains the following members:

- 3 `private` instance variables `hour`, `minute`, and `second`.
- Constructors, getters and setters.
- A method `setTime()` to set `hour`, `minute` and `second`.
- A `toString()` that returns "`hh:mm:ss`" with leading zero if applicable.
- A method `nextSecond()` that advances `this` instance by one second. It returns `this` instance to support chaining (cascading) operations, e.g., `t1.nextSecond().nextSecond()`. Take note that the `nextSecond()` of `23:59:59` is `00:00:00`.

Write the `Time` class and a test driver to test all the public methods. No input validations are required.

The `Time` Class (`Time.java`)

```
/**
 * The Time class models a time instance with second, minute and hour.
 * This class does not perform input validation for second, minute and hour.
 */
public class Time {
   // The private instance variables
   private int second, minute, hour;

   // The constructors (overloaded)
   /** Constructs a Time instance with the given second, minute and hour. No input validation */
   public Time(int second, int minute, int hour) {
      this.second = second;
      this.minute = minute;
      this.hour = hour;
   }
   /** Constructs a Time instance with the default values */
   public Time() {  // the default constructor
      this.second = 0;
      this.minute = 0;
      this.hour = 0;
   }

   // The public getters/setters for the private variables.
   /** Returns the second */
   public int getSecond() {
      return this.second;
   }
   /** Returns the minute */
   public int getMinute() {
      return this.minute;
   }
   /** Returns the hour */
   public int getHour() {
      return this.hour;
```

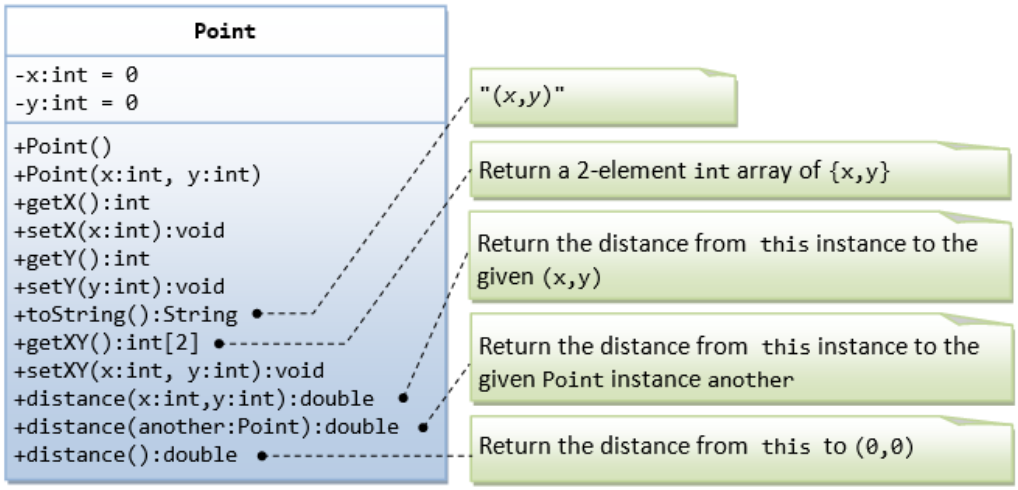
```
   }
   /** Sets the second. No input validation */
   public void setSecond(int second) {
      this.second = second;
   }
   /** Sets the minute. No input validation */
   public void setMinute(int minute) {
      this.minute = minute;
   }
   /** Sets the hour. No input validation */
   public void setHour(int hour) {
      this.hour = hour;
   }

   /** Returns a self-descriptive string in the form of  "hh:mm:ss" with leading zeros */
   public String toString() {
        // Use built-in function String.format() to form a formatted String
        return String.format("%02d:%02d:%02d", hour, minute, second);
              // Specifier "0" to print leading zeros, if available.
   }

   /** Sets second, minute and hour to the given values */
   public void setTime(int second, int minute, int hour) {
      // No input validation
      this.second = second;
      this.minute = minute;
      this.hour = hour;
   }

   /** Advances this Time instance by one second, and returns this instance to support chaining */
   public Time nextSecond() {
      ++second;
      if (second >= 60) {
         second = 0;
         ++minute;
         if (minute >= 60) {
            minute = 0;
            ++hour;
            if (hour >= 24) {
               hour = 0;
            }
         }
      }
      return this;   // Return "this" instance, to support chaining operations
                     // e.g., t1.nextSecond().nextSecond()
   }
}
```

A Test Driver (`TestTime.java`)

```
/**
 * A Test Driver for the Time class
 */
public class TestTime {
   public static void main(String[] args) {
      // Test Constructors and toString()
      Time t1 = new Time(1, 2, 3);
      System.out.println(t1);  // toString()
      //03:02:01
      Time t2 = new Time();    // The default constructor
      System.out.println(t2);
      //00:00:00

      // Test Setters and Getters
      t1.setHour(4);
      t1.setMinute(5);
      t1.setSecond(6);
      System.out.println(t1);  // run toString() to inspect the modified instance
      //04:05:06
      System.out.println("Hour is: " + t1.getHour());
      //Hour is: 4
      System.out.println("Minute is: " + t1.getMinute());
      //Minute is: 5
      System.out.println("Second is: " + t1.getSecond());
      //Second is: 6
```

```
      // Test setTime()
      t1.setTime(58, 59, 23);
      System.out.println(t1);
      //23:59:58

      // Test nextSecond() and chaining
      System.out.println(t1.nextSecond()); // Return an instance of Time. Invoke Time's toString()
      //23:59:59
      System.out.println(t1.nextSecond().nextSecond().nextSecond());  // chaining
      //00:00:02
   }
}
```

3.3 EG. 3: The `Point` class



A `Point` class models a 2D point at `(x,y)`, as shown in the class diagram. It contains the following members:

- 2 `private` instance variables x and y, which maintain the location of the point.
- Constructors, getters and setters.
- A method `setXY()`, which sets the x and y of the point; and a method `getXY()`, which returns the x and y in a 2-element `int` array.
- A `toString()`, which returns "`(x,y)`".
- 3 versions of overloaded `distance()`:
  - `distance(int x, int y)` returns the distance from `this` instance to the given point at `(x,y)`.
  - `distance(Point another)` returns the distance from `this` instance to the given `Point` instance (called `another`).
  - `distance()` returns the distance from `this` instance to `(0,0)`.

The `Point` Class (`Point.java`)

```
/**
 * The Point class models a 2D point at (x, y).
 */
public class Point {
   // The private instance variables
   private int x, y;

   // The constructors (overloaded)
   /** Construct a Point instance with the default values */
   public Point() {  // The default constructor
      this.x = 0;
      this.y = 0;
   }
   /** Construct a Point instance with the given x and y values */
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // The public getters and setters
   /** Returns the value of x */
   public int getX() {
      return this.x;
```

```
   }
   /** Sets the value of x */
   public void setX(int x) {
      this.x = x;
   }
   /** Returns the value of y */
   public int getY() {
      return this.y;
   }
   /** Sets the value of y */
   public void setY(int y) {
      this.y = y;
   }

   /** Returns a self-descriptive string in the form of "(x,y)" */
   public String toString() {
      return "(" + this.x + "," + this.y + ")";
   }

   /** Returns a 2-element int array containing x and y */
   public int[] getXY() {
      int[] results = new int[2];
      results[0] = this.x;
      results[1] = this.y;
      return results;
   }

   /** Sets both x and y */
   public void setXY(int x, int y) {
      this.x = x;
      this.y = y;
   }

   /** Return the distance from this instance to the given point at (x,y). Invoke via p1.distance(1,2) */
   public double distance(int x, int y) {
      int xDiff = this.x - x;
      int yDiff = this.y - y;
      return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
   }
   /** Returns the distance from this instance to the given Point instance. Invoke via p1.distance(p2) */
   public double distance(Point another) {
      int xDiff = this.x - another.x;
      int yDiff = this.y - another.y;
      return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
   }
   /** Returns the distance from this instance to (0,0). Invoke via p1.distance() */
   public double distance() {
      return Math.sqrt(this.x*this.x + this.y*this.y);
   }
}
```

A Test Driver (`TestPoint.java`)

```
/**
 * A Test Driver for the Point class.
 */
public class TestPoint {
   public static void main(String[] args) {
      // Test constructors and toString()
      Point p1 = new Point(1, 2);
      System.out.println(p1);  // toString()
      //(1,2)
      Point p2 = new Point();  // default constructor
      System.out.println(p2);
      //(0,0)

      // Test Setters and Getters
      p1.setX(3);
      p1.setY(4);
      System.out.println(p1);  // run toString() to inspect the modified instance
      //(3,4)
      System.out.println("X is: " + p1.getX());
      //X is: 3
      System.out.println("Y is: " + p1.getY());
      //Y is: 4
```

```
      // Test setXY() and getXY()
      p1.setXY(5, 6);
      System.out.println(p1);  // toString()
      //(5,6)
      System.out.println("X is: " + p1.getXY()[0]);
      //X is: 5
      System.out.println("Y is: " + p1.getXY()[1]);
      //Y is: 6

      // Test the 3 overloaded versions of distance()
      p2.setXY(10, 11);
      System.out.printf("Distance is: %.2f%n", p1.distance(10, 11));
      //Distance is: 7.07
      System.out.printf("Distance is: %.2f%n", p1.distance(p2));
      //Distance is: 7.07
      System.out.printf("Distance is: %.2f%n", p2.distance(p1));
      //Distance is: 7.07
      System.out.printf("Distance is: %.2f%n", p1.distance());
      //Distance is: 7.81
   }
}
```

3.4 EG. 4: The `Time` class with Input Validation

In this example, we shall validate the inputs to ensure that `0≤hour≤23`, `0≤minute≤59`, and `0≤second≤59`. We re-write our `Time` class as follows. Take note that all the validations are done in the setters. All other methods (such as constructors and `setTime()` invoke the setters to perform input validations - so as to avoid duplication of codes.

```
/**
 * The Time class models a time instance with second, minute and hour.
 * This class performs input validations.
 */
public class Time {
   // The private instance variables - with input validations.
   private int second;  // valid range is [0, 59]
   private int minute;  // valid range is [0, 59]
   private int hour;    // valid range is [0, 23]

   // Input validations are done in the setters.
   // All the other methods (such as constructors and setTime()) invoke
   //   these setters to perform input validations to avoid code duplication.
   /** Sets the second to the given value with input validation */
   public void setSecond(int second) {
      if (second >=0 && second <= 59) {
         this.second = second;
      } else {
         this.second = 0;  // Set to 0 and print error message
         System.out.println("error: invalid second");
      }
   }
   /** Sets the minute to the given value with input validation */
   public void setMinute(int minute) {
      if (minute >=0 && minute <= 59) {
         this.minute = minute;
      } else {
         this.minute = 0;
         System.out.println("error: invalid minute");
      }
   }
   /** Sets the hour to the given value with input validation */
   public void setHour(int hour) {
      if (hour >=0 && hour <= 23) {
         this.hour = hour;
      } else {
         this.hour = 0;
         System.out.println("error: invalid hour");
      }
   }

   /** Sets second, minute and hour to the given values with input validation */
   public void setTime(int second, int minute, int hour) {
      // Invoke setters to do input validation
      this.setSecond(second);
      this.setMinute(minute);
      this.setHour(hour);
```

```
   }

   /** Constructs a Time instance with the given values with input validation */
   public Time(int second, int minute, int hour) {
      // Invoke setters to do input validation
      this.setTime(second, minute, hour);
   }
   /** Constructs a Time instance with default values */
   public Time() {  // The default constructor
      this.second = 0;
      this.minute = 0;
      this.hour = 0;
   }

   // The public getters
   /** Returns the second */
   public int getSecond() {
      return this.second;
   }
   /** Returns the minute */
   public int getMinute() {
      return this.minute;
   }
   /** Returns the hour */
   public int getHour() {
      return this.hour;
   }

   /** Returns a self-descriptive string in the form of "hh:mm:ss" with leading zeros */
   public String toString() {
        return String.format("%02d:%02d:%02d", hour, minute, second);
   }
   /** Advances this Time instance by one second and returns this instance to support chaining */
   public Time nextSecond() {
      ++second;
      if (second == 60) {  // We are sure that second <= 60 here because of the input validation
         second = 0;
         ++minute;
         if (minute == 60) {
            minute = 0;
            ++hour;
            if (hour == 24) {
               hour = 0;
            }
         }
      }
      return this;   // Return this instance, to support chaining
   }
}
```

Exercise: Write a Test Driver to test the above `Time` class with various valid and invalid values.

3.5 EG. 5 (Advanced): The `Time` Class with Input Validation via Exception Handling

In the previous example, we print a error message and set the variable to 0, if the input is invalid. This is less than perfect. The proper way to handle invalid inputs is via the so-called *exception handling* mechanism.

The revised `Time.java` that uses exception handling mechanism is as follows:

```
1  /**
2   * The Time class models a time instance with second, minute and hour.
3   * This class performs input validations using exception handling.
4   */
5  public class Time {
6     // The private instance variables - with input validations.
7     private int second;  // valid range is [0, 59]
8     private int minute;  // valid range is [0, 59]
9     private int hour;    // valid range is [0, 23]
10
11    // Input validations are done in the setters.
12    // All the other methods (such as constructors and setTime()) invoke
13    //   these setters to perform input validations to avoid code duplication.
14    /** Sets the second to the given value with input validation */
15    public void setSecond(int second) {
```

```
16          if (second >=0 && second <= 59) {
17             this.second = second;
18          } else {
19             throw new IllegalArgumentException("invalid second");
20          }
21       }
22       /** Sets the minute to the given value with input validation */
23       public void setMinute(int minute) {
24          if (minute >=0 && minute <= 59) {
25             this.minute = minute;
26          } else {
27             throw new IllegalArgumentException("invalid minute");
28          }
29       }
30       /** Sets the hour to the given value with input validation */
31       public void setHour(int hour) {
32          if (hour >=0 && hour <= 23) {
33             this.hour = hour;
34          } else {
35             throw new IllegalArgumentException("invalid hour");
36          }
37       }
38
39       /** Sets second, minute and hour to the given values with input validation */
40       public void setTime(int second, int minute, int hour) {
41          // Invoke setters to do input validation
42          this.setSecond(second);
43          this.setMinute(minute);
44          this.setHour(hour);
45       }
46
47       /** Constructs a Time instance with the given values with input validation */
48       public Time(int second, int minute, int hour) {
49          // Invoke setters to do input validation
50          this.setTime(second, minute, hour);
51       }
52       /** Constructs a Time instance with default values */
53       public Time() {  // The default constructor
54          this.second = 0;
55          this.minute = 0;
56          this.hour = 0;
57       }
58
59       // The public getters
60       /** Returns the second */
61       public int getSecond() {
62          return this.second;
63       }
64       /** Returns the minute */
65       public int getMinute() {
66          return this.minute;
67       }
68       /** Returns the hour */
69       public int getHour() {
70          return this.hour;
71       }
72
73       /** Returns a self-descriptive string in the form of "hh:mm:ss" with leading zeros */
74       public String toString() {
75            return String.format("%02d:%02d:%02d", hour, minute, second);
76       }
77       /** Advances this Time instance by one second and returns this instance to support chaining */
78       public Time nextSecond() {
79          ++second;
80          if (second == 60) {  // We are sure that second <= 60 here because of the input validation
81             second = 0;
82             ++minute;
83             if (minute == 60) {
84                minute = 0;
85                ++hour;
86                if (hour == 24) {
```

```
                hour = 0;
             }
          }
       }
       return this;   // Return this instance, to support chaining
    }
}
```

Exception Handling

What to do if an invalid `hour`, `minute` or `second` was given as input argument? Print an error message? Terminate the program abruptly? Continue operation by setting the parameter to its default? This is a really hard decision and there is no perfect solution that suits all situations.

In Java, instead of printing an error message, you can throw an so-called `Exception` object (such as `IllegalArgumentException`) to the caller, and let the caller handles the exception gracefully.

For example, we validate the `hour` input as follows:

```
// Throw an exception if input is invalid
public void setHour(int hour) {
   if (hour >= 0 && hour <= 23) {
      this.hour = hour;
   } else {
      throw new IllegalArgumentException("invalid hour");
   }
}
```

If the caller provides an invalid `hour` without handling the exception, the program terminates with a runtime error. For example,

```
Time t11 = new Time(58, 59, 24);
//java.lang.IllegalArgumentException: invalid hour
```

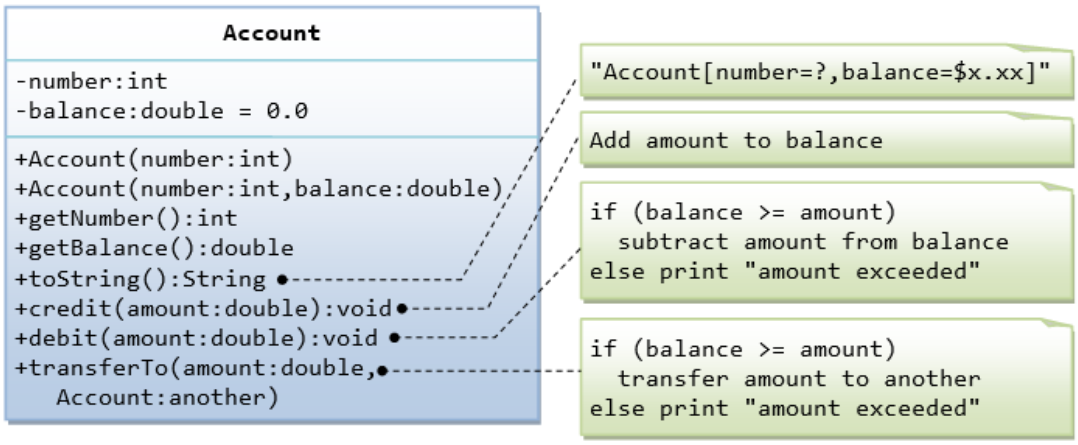
The caller can choose to handle the exception using the `try-catch` construct to process the exception *gracefully*. For example,

```
 /**
  * A Test Driver for the Time class
  */
public class TestTime {
   public static void main(String[] args) {
      // Case 1: valid input
      //int hour = 23, minute = 58, second = 58;
      // Case 2: invalid input
      int hour = 24, minute = 58, second = 58;
      Time t12;

      //t12 = new Time(second, minute, hour);
      // Without try-catch, the program will terminate abruptly if exception thrown

      try {
         t12 = new Time(second, minute, hour);
         // If input is invalid, throw exception. Skip the rest, goto "catch".
         // Else complete "try", skip "catch"
         System.out.println("valid input");
      } catch (IllegalArgumentException ex) {
         // You have the opportunity to do something to recover from the error.
         ex.printStackTrace();  // print error messages
         System.out.println("Error in input. Set to default value");
         // You should ask the user to provide the valid input instead!
         t12 = new Time();
      }

      System.out.println("Time is " + t12);
      System.out.println("Life goes on...");

      // Case 1 output
      //valid input
      //Time is 23:58:58
      //Life goes on...

      // Case 2 output
      //java.lang.IllegalArgumentException: invalid hour
      //Error in input. Set to default value
      //Time is 00:00:00
      //Life goes on...
```

```
    }
}
```

The statements in the `try`-clause will be executed. If all the statements in the `try`-clause are successful, the `catch`-clause is ignored, and execution continues to the next statement after `try-catch`. However, if one of the statement in the `try`-clause throws an exception (in this case, an `IllegalArgumentException`), the rest of `try`-clause will be skipped, and the execution will be transferred to the `catch`-clause. The program always continues to the next statement after the `try-catch` (instead of abruptly terminated).

3.6 EG. 6: The `Account` Class



A class called `Account`, which models a bank account, is designed as shown in the class diagram. It contains the following members:

- Two `private` instance variables: `accountNumber` (`int`), and `balance` (`double`) which maintains the current account balance.
- Constructors (overloaded).
- Getters and Setters for the `private` instance variables. There is no setter for `accountNumber` as it is not designed to be changed.
- `public` methods `credit()` and `debit()`, which adds/subtracts the given `amount` to/from the balance, respectively.
- A `toString()`, which returns "`A/C no:xxx, Balance=$xxx.xx`", with `balance` rounded to two decimal places.

Write the `Account` class and a test driver to test all the `public` methods.

The `Account` Class (`Account.java`)

```
/**
 * The Account class models a bank account with a balance.
 */
public class Account {
   // The private instance variables
   private int number;
   private double balance;

   // The constructors (overloaded)
   /** Constructs an Account instance with the given number and initial balance of 0 */
   public Account(int number) {
      this.number = number;
      this.balance = 0.0;  // "this." is optional
   }
   /** Constructs an Account instance with the given number and initial balance */
   public Account(int number, double balance) {
      this.number = number;
      this.balance = balance;
   }

   // The public getters/setters for the private instance variables.
   // No setter for number because it is not designed to be changed.
   // No setter for balance as it is changed via credit() and debit()
   /** Returns the number */
   public int getNumber() {
      return this.number;  // "this." is optional
   }
   /** Returns the balance */
   public double getBalance() {
      return this.balance;  // "this." is optional
   }
```

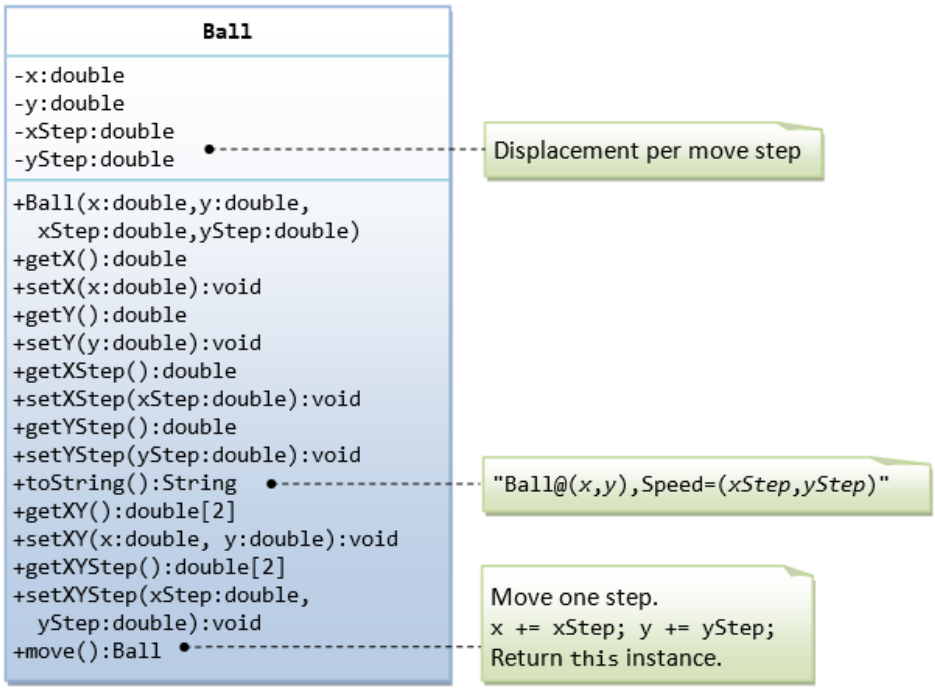
```
32
33      /** Returns a string description of this instance */
34      public String toString() {
35         // Use built-in function System.format() to form a formatted String
36         return String.format("Account[number=%d,balance=$%.2f]", number, balance);
37      }
38
39      /** Add the given amount to the balance */
40      public void credit(double amount) {
41         balance += amount;
42      }
43
44      /** Subtract the given amount from balance, if balance >= amount */
45      public void debit(double amount) {
46         if (balance >= amount) {
47            balance -= amount;
48         } else {
49            System.out.println("amount exceeded");
50         }
51      }
52
53      /** Transfer the given amount to Account another, if balance >= amount */
54      public void transferTo(double amount, Account another) {
55         if (balance >= amount) {
56            this.balance -= amount;
57            another.balance += amount;
58         } else {
59            System.out.println("amount exceeded");
60         }
61      }
62   }
```

A Test Driver for the `Account` Class (`TestAccount.java`)

```
/**
 * A Test Driver for the Account class.
 */
public class TestAccount {
   public static void main(String[] args) {
      // Test Constructors and toString()
      Account a1 = new Account(5566);
      System.out.println(a1);
      //Account[number=5566,balance=$0.00]
      Account a2 = new Account(1234, 99.9);
      System.out.println(a2);
      //Account[number=1234,balance=$99.90]

      // Test getters
      System.out.println("The account Number is: " + a2.getNumber());
      //The account Number is: 1234
      System.out.println("The balance is: " + a2.getBalance());
      //The balance is: 99.9

      // Test credit(), debit() and transferTo()
      a1.credit(11.1);
      System.out.println(a1);
      //Account[number=5566,balance=$11.10]
      a1.debit(5.5);
      System.out.println(a1);
      //Account[number=5566,balance=$5.60]
      a1.debit(500);   // Test debit() error
      //amount exceeded
      System.out.println(a1);
      //Account[number=5566,balance=$5.60]

      a2.transferTo(1.0, a1);
      System.out.println(a1);
      //Account[number=5566,balance=$6.60]
      System.out.println(a2);
      //Account[number=1234,balance=$98.90]
    }
}
```

Try: Re-design the methods `credit()`, `debit()` and `transferTo()` to return `this` instance, so that these methods can be chained, e.g., `a1.credit(10).credit(20).debit(5).transferTo(5.5, a2)`. For `debit()` and `transferTo()` you need to throw an exception instead printing an error message. See the above `Time` class.

3.7 EG. 7: The `Ball` class



A `Ball` class models a moving ball, is designed as shown in the class diagram. It contains the following members:

- 4 `private` variables x, y, `xStep`, `yStep`, which maintain the position of the ball and the displacement per move step.
- Constructors, getters and setters.
- Method `setXY()` and `setXYStep()`, which sets the position and step size of the ball; and `getXY()` and `getXYSpeed()`.
- A `toString()`, which returns "`Ball@(x,y),speed=(xStep,yStep)`".
- A method `move()`, which increases `x` and `y` by `xStep` and `yStep` respectively; and returns `this` instance to support chaining operation.

The `Ball` Class (`Ball.java`)

```
/**
 * The Ball class models a moving ball at (x, y) with displacement
 *   per move-step of (xStep, yStep).
 */
public class Ball {
   // The private instance variables
   private double x, y, xStep, yStep;

   /** Constructs a Ball instance with the given input */
   public Ball(double x, double y, double xStep, double yStep) {
      this.x = x;
      this.y = y;
      this.xStep = xStep;
      this.yStep = yStep;
   }

   // The public getters and setters for the private variables
   public double getX() {
      return this.x;
   }
   public void setX(double x) {
      this.x = x;
   }
   public double getY() {
      return this.y;
   }
   public void setY(double y) {
      this.y = y;
```

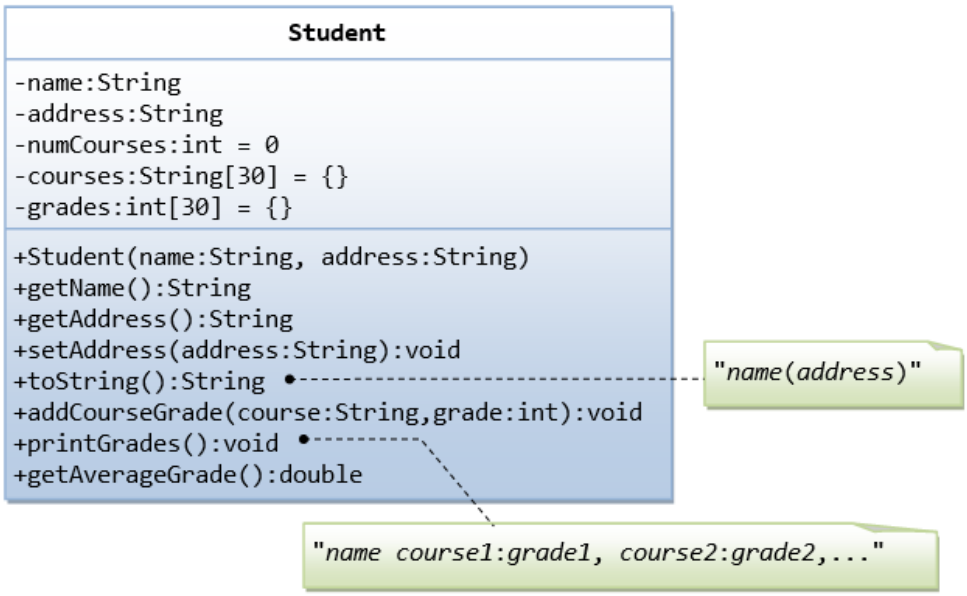
```
   }
   public double getXStep() {
      return this.xStep;
   }
   public void setXStep(double xStep) {
      this.xStep = xStep;
   }
   public double getYStep() {
      return this.yStep;
   }
   public void setYStep(double yStep) {
      this.yStep = yStep;
   }

   /** Returns a self-descriptive String */
   public String toString() {
      return "Ball@(" + x + "," + y + "),speed=(" + xStep + "," + yStep + ")";
   }

   /** Returns the x and y position in a 2-element double array */
   public double[] getXY() {
      double[] results = new double[2];
      results[0] = this.x;
      results[1] = this.y;
      return results;
   }
   /** Sets the x and y position */
   public void setXY(double x, double y) {
      this.x = x;
      this.y = y;
   }
   /** Returns the xStep and yStep in a 2-element double array */
   public double[] getXYStep() {
      double[] results = new double[2];
      results[0] = this.xStep;
      results[1] = this.yStep;
      return results;
   }
   /** Sets the xStep and yStep */
   public void setXYStep(double xStep, double yStep) {
      this.xStep = xStep;
      this.yStep = yStep;
   }

   /** Moves a step by increment x and y by xStep and yStep, respectively.
       Return "this" instance to support chaining operation. */
   public Ball move() {
      x += xStep;
      y += yStep;
      return this;
   }
}
```

A Test Driver (`TestBall.java`)

```
import java.util.Arrays;
/**
 * A Test Driver for the Ball class.
 */
public class TestBall {
   public static void main(String[] args) {
      // Test constructor and toString()
      Ball b1 = new Ball(1, 2, 11, 12);
      System.out.println(b1);  // toString()
      //Ball@(1.0,2.0),speed=(11.0,12.0)

      // Test Setters and Getters
      b1.setX(3);
      b1.setY(4);
      b1.setXStep(13);
      b1.setYStep(14);
      System.out.println(b1);
      //Ball@(3.0,4.0),speed=(13.0,14.0)
      System.out.println("x is: " + b1.getX());
      //x is: 3.0
      System.out.println("y is: " + b1.getY());
```

```
      //y is: 4.0
      System.out.println("xStep is: " + b1.getXStep());
      //xStep is: 13.0
      System.out.println("yStep is: " + b1.getYStep());
      //yStep is: 14.0

      // Test setXY(), getXY(), setXYStep(), getXYStep()
      b1.setXY(5, 6);
      b1.setXYStep(15, 16);
      System.out.println(b1);  // toString()
      //Ball@(5.0,6.0),speed=(15.0,16.0)
      System.out.println("x and y are: " + Arrays.toString(b1.getXY()));  // use utility to print array
      //x and y are: [5.0, 6.0]
      System.out.println("xStep and yStep are: " + Arrays.toString(b1.getXYStep()));
      //xStep and yStep are: [15.0, 16.0]

      // Test move() and chaining
      System.out.println(b1.move());  // toString()
      //Ball@(20.0,22.0),speed=(15.0,16.0)
      System.out.println(b1.move().move().move());
      //Ball@(65.0,70.0),speed=(15.0,16.0)
   }
}
```

3.8 EG. 8: The `Student` Class



Suppose that our application requires us to model students. A student has a name and an address. We are required to keep track of the courses taken by each student, together with the grades (between 0 and 100) for each of the courses. A student shall not take more than 30 courses for the entire program. We are required to print all course grades, and also the overall average grade.

We can design the `Student` class as shown in the class diagram. It contains the following members:

- `private` instance variables `name` (`String`), `address` (`String`), `numCourses` (`int`), `course` (`String[30]`) and `grades` (`int[30]`). The `numCourses` keeps track of the number of courses taken by this student so far. The `courses` and `grades` are two parallel arrays, storing the courses taken (e.g., `{"IM101", "IM102", "IM103"}`) and their respective grades (e.g. `{89, 56, 98}`).
- A constructor that constructs an instance with the given `name` and `Address`. It also constructs the `courses` and `grades` arrays and set the `numCourses` to 0.
- Getters for `name` and `address`; setter for `address`. No setter is defined for `name` as it is not designed to be changed.
- A `toString()`, which prints "`name(address)`".
- A method `addCourseGrade(course, grade)`, which appends the given `course` and `grade` into the `courses` and `grades` arrays, respectively; and increments `numCourses`.
- A method `printGrades()`, which prints "`name course1:grade1, course2:grade2,...`".
- A method `getAverageGrade()`, which returns the average grade of all the courses taken.

The `Student` Class (`Student.java`)

```
1  /**
```

```
 * The student class models a student having courses and grades.
 */
public class Student {
   // The private instance variables
   private String name;
   private String address;
   // The courses taken and grades for the courses are kept in 2 parallel arrays
   private String[] courses;
   private int[] grades;     // valid range is [0, 100]
   private int numCourses;   // Number of courses taken so far
   private static final int MAX_COURSES = 30;  // Maximum number of courses taken by student

   /** Constructs a Student instance with the given input */
   public Student(String name, String address) {
      this.name = name;
      this.address = address;
      courses = new String[MAX_COURSES];  // allocate arrays
      grades = new int[MAX_COURSES];
      numCourses = 0;                     // no courses so far
   }

   // The public getters and setters.
   // No setter for name as it is not designed to be changed.
   /** Returns the name */
   public String getName() {
      return this.name;
   }
   /** Returns the address */
   public String getAddress() {
      return this.address;
   }
   /** Sets the address */
   public void setAddress(String address) {
      this.address = address;
   }

   /** Returns a self-descriptive String */
   public String toString() {
      return name + "(" + address + ")";
   }

   /** Adds a course and grade */
   public void addCourseGrade(String course, int grade) {
      courses[numCourses] = course;
      grades[numCourses] = grade;
      ++numCourses;
   }

   /** Prints all courses and their grades */
   public void printGrades() {
      System.out.print(name);
      for (int i = 0; i < numCourses; ++i) {
         System.out.print(" " + courses[i] + ":" + grades[i]);
      }
      System.out.println();
   }

   /** Computes the average grade */
   public double getAverageGrade() {
      int sum = 0;
      for (int i = 0; i < numCourses; ++i) {
         sum += grades[i];
      }
      return (double)sum/numCourses;
   }
}
```

A Test Driver for the `Student` Class (`TestStudent.java`)

```
/**
 * A test driver program for the Student class.
```

```
 */
public class TestStudent {
   public static void main(String[] args) {
      // Test constructor and toString()
      Student ahTeck = new Student("Tan Ah Teck", "1 Happy Ave");
      System.out.println(ahTeck);  // toString()
      //Tan Ah Teck(1 Happy Ave)

      // Test Setters and Getters
      ahTeck.setAddress("8 Kg Java");
      System.out.println(ahTeck);
      //Tan Ah Teck(8 Kg Java)
      System.out.println(ahTeck.getName());
      //Tan Ah Teck
      System.out.println(ahTeck.getAddress());
      //8 Kg Java

      // Test addCourseGrade(), printGrades() and getAverageGrade()
      ahTeck.addCourseGrade("IM101", 89);
      ahTeck.addCourseGrade("IM102", 57);
      ahTeck.addCourseGrade("IM103", 96);
      ahTeck.printGrades();
      //Tan Ah Teck IM101:89 IM102:57 IM103:96
      System.out.printf("The average grade is %.2f%n", ahTeck.getAverageGrade());
      //The average grade is 80.67
   }
}
```

Notes: We used arrays In this example, which has several limitations. Arrays need to be pre-allocated with a fixed-length. Furthermore, we need two parallel arrays to keep track of two entities. There are advanced data structures that could represent these data better and more efficiently.

3.9 Exercises

LINK TO EXERCISES

LINK TO JAVA REFERENCES & RESOURCES

Latest version tested: JDK 1.13.1
Last modified: February, 2020