Java Programming Tutorial
Object-oriented Programming (OOP) Basics

TABLE OF CONTENTS

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 motherboard is a 4-layer or 6-layer board, whether the hard disk has 4 or 6 plates; 3 inches or 5 inches in diameter, whether 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, break, transmission, etc. 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 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 60 years ago, we have written tons and tons of programs. However, for each new application, we have to re-invent the wheels and write the program from scratch.

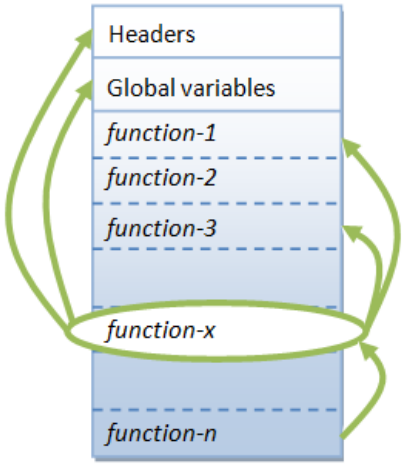
Why re-invent the wheels?

Can we do this in traditional procedural-oriented programming language such as C, Fortran, Cobol, or Pascal?

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

1. The programs are made up of functions. Functions are often not *reusable*. It is very difficult to copy a function from one program and reuse in another program because the the function is likely to reference the headers, 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. (Imagine using assembly codes, which is a very low level code, to write a computer soccer game. C is better but no much better.)

In brief, the traditional procedural-languages *separate* the data structures and algorithms of the software entities.



A function (in C) is not well-encapsulated

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:

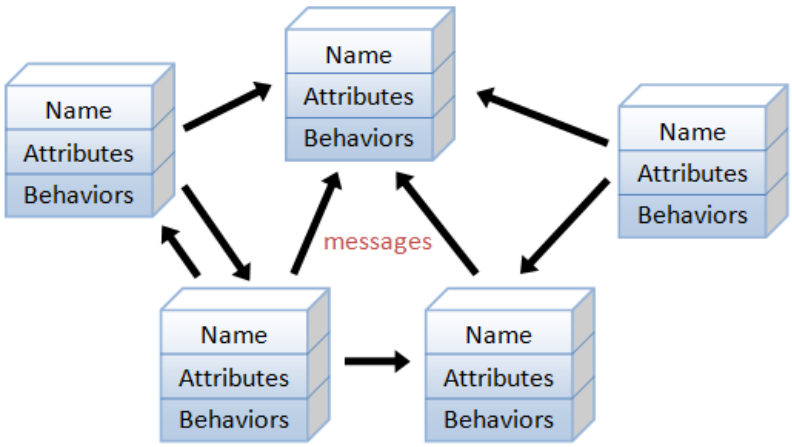
1. 80% of the budget went to the software (while the remaining 20% to the hardware).
2. More than 80% of the software budget went to maintenance (only the remaining 20% for new software development).
3. Hardware components could be applied to various products, and their integrity normally did not affect 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 (OOP) languages are designed to overcome these problems.

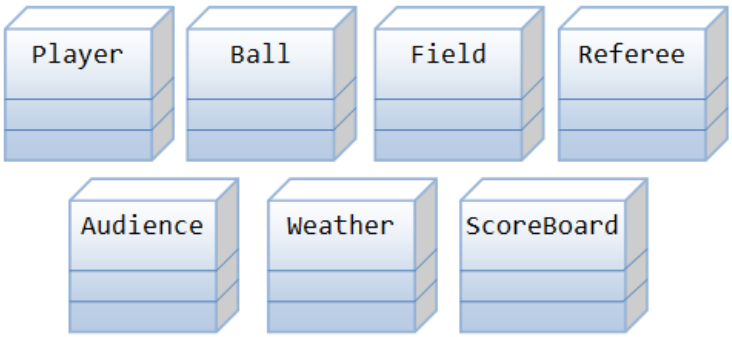
1. The basic unit of OOP is a *class*, which encapsulates both the *static attributes* and *dynamic behaviors* within a "box", and specifies the public interface for using these boxes. Since the class is well-encapsulated (compared with the function), it is easier to reuse these classes. In other words, 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++, 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.



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

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.

- 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:



Classes (Entities) in a Computer Soccer Game

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.

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.

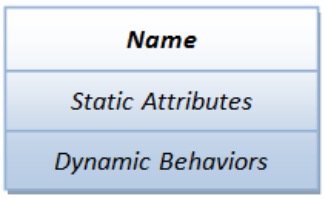
An *instance* is *a realization of a particular item of a class*. In other words, an instance is an *instantiation* of a class. All the instances of a class have similar properties, as described in the class definition. For example, you can define a class called "`Student`" and create three instances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

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

2.2 A Class is a 3-Compartment Box

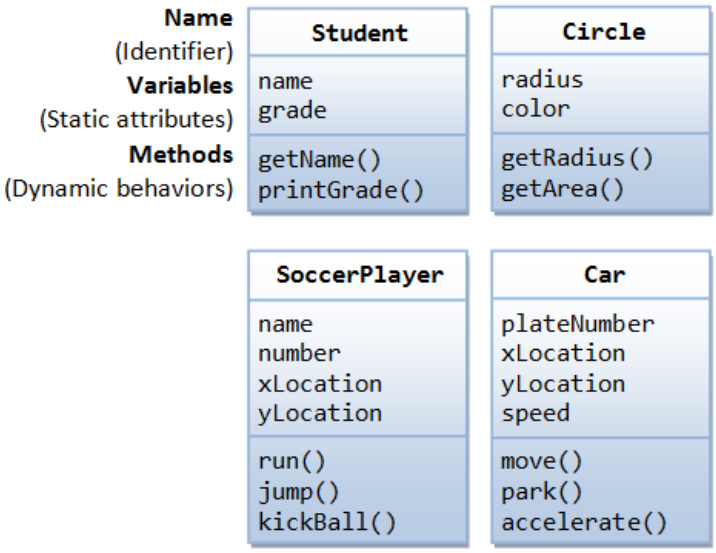
A class can be visualized as a three-compartment box, as illustrated:

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.



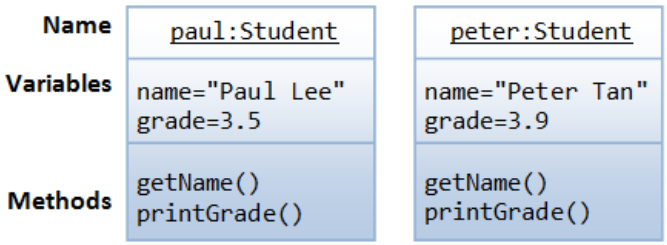
A class is a 3-compartment box

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 "`perter`".



Two instances 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 method, respectively. Class name is shown in bold and centralized. An instance (object) is also represented as a 3-compartment box, with instance name shown as *`instanceName:Classname`* and underlined.

Brief Summary

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

2.3 Class Definition

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 xLocation, yLocation;
```

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

The syntax for class definition in Java is:

```
[AccessControlModifier] class ClassName {
   // class body contains definition of 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;
// Allocate and 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();
```

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. reference the method or variable via the *dot operator*.

For example, suppose that we have a class called `Circle`, with two variables (`radius` and `color`) and two 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()`, to invoke the `getArea()` method of instance `c2`.

For example,

```
// 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).

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). A member variable can also be an instance of a certain class (to be discussed later).

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`. Take note that variable name begins with an lowercase, while class name begins with an uppercase.

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 parameters 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 ([argumentList]) {
   // method body or implementation
   ......
}
```

For examples:

```
public double getArea() {
   return radius*radius*Math.PI;
}
```

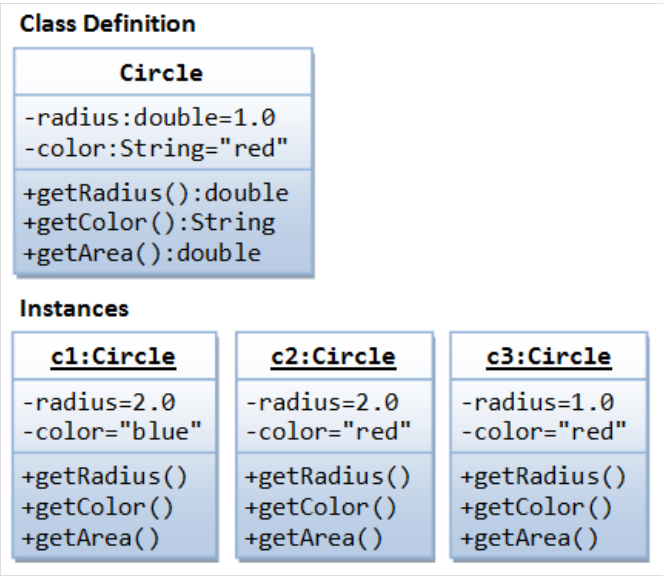
Method Naming Convention: A method name shall be a verb, or a verb phrase made up of several words. The first word is in lowercase and the rest of the words are initial-capitalized (camel-case). For example, `getRadius()`, `getParameterValues()`.

Take note that variable name is a noun (denoting a static attribute), while method name is a verb (denoting an action). They have the same naming convention. Nevertheless, you can easily distinguish them from the context. Methods take arguments in parentheses (possibly zero argument with empty parentheses), but variables do not. In this writing, methods are denoted with a pair of parentheses, e.g., `println()`, `getArea()` for clarity.

2.8 Putting them together: An OOP Example

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

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




```
1  // Define the Circle class
2  public class Circle {    // Save as "Circle.java"
3     // Private variables
4     private double radius;
5     private String color;
6
7     // Constructors (overloaded)
8     public Circle() {                  // 1st Constructor
9        radius = 1.0;
10       color = "red";
11    }
12    public Circle(double r) {          // 2nd Constructor
13       radius = r;
14       color = "red";
15    }
16    public Circle(double r, String c) { // 3rd Constructor
```

```
      radius = r;
      color = c;
   }

   // Public methods
   public double getRadius() {
      return radius;
   }
   public String getColor() {
      return color;
   }
   public double getArea() {
      return radius*radius*Math.PI;
   }
}
```

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 and used in other programs.

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

```
// Test driver program for the Circle class
public class TestCircle {    // Save as "TestCircle.java"
   public static void main(String[] args) {   // Execution entry point
      // Construct an instance of the Circle class called c1
      Circle c1 = new Circle(2.0, "blue");    // Use 3rd constructor
      System.out.println("Radius is " + c1.getRadius()  // use dot operator to invoke member methods
         + " Color is " + c1.getColor()
         + " Area is " + c1.getArea());

      // Construct another instance of the Circle class called c2
      Circle c2 = new Circle(2.0);          // Use 2nd constructor
      System.out.println("Radius is " + c2.getRadius()
         + " Color is " + c2.getColor()
         + " Area is " + c2.getArea());

      // Construct yet another instance of the Circle class called c3
      Circle c3 = new Circle();             // Use 1st constructor
      System.out.println("Radius is " + c3.getRadius()
         + " Color is " + c3.getColor()
         + " Area is " + c3.getArea());
   }
}
```

Compile `TestCircle.java` into `TestCircle.class`. Run the `TestCircle` and study the output:

```
Radius is 2.0 Color is blue Area is 12.566370614359172
Radius is 2.0 Color is red Area is 12.566370614359172
Radius is 1.0 Color is red Area is 3.141592653589793
```

2.9 Constructors

A *constructor* is 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 create 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, and by classname convention, begins with an uppercase.
- Constructor has no return type (or implicitly returns `void`). Hence, 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. You cannot call the constructor afterwards.
- Constructors are not inherited (to be explained later).

A constructor with no parameter is called the *default constructor*, which initializes the member variables to their default value. For example, the `Circle()` in the above example.

2.10 Method Overloading

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

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

```
// Example to illustrate Method Overloading
public class TestMethodOverloading {
   public static int average(int n1, int n2) {          // A
      return (n1+n2)/2;
   }

   public static double average(double n1, double n2) { // B
      return (n1+n2)/2;
   }

   public static int average(int n1, int n2, int n3) {  // C
      return (n1+n2+n3)/3;
   }

   public static void main(String[] args) {
      System.out.println(average(1, 2));     // Use A
      System.out.println(average(1.0, 2.0)); // Use B
      System.out.println(average(1, 2, 3));  // Use C
      System.out.println(average(1.0, 2));   // Use B - int 2 implicitly casted to double 2.0
      // average(1, 2, 3, 4); // Compilation Error - No matching method
   }
}
```

Overloaded Constructors for a Class

The above `Circle` class has three 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 inside 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`, and re-run the 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 notation, `public` members are denoted with a "+", while `private` members with a "-" in the class diagram.

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 `setRadius()`.

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 Getters and Setters

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

To allow other classes to *modify* the value of a `private` variable says `xxx`, you shall provide a *set method* (or *setter* or *mutator method*) called `setXxx()`. A set method could provide data validation (such as range checking), and 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 available within the `Circle` class and not visible in any other classes - including `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/she has to provide the appropriate set methods (or setters or mutator methods), e.g.,

```
// Setter for color
public void setColor(String c) {
   color = c;
}

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

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;
         // "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 certainly causes naming conflict. To avoid the naming conflict, you could name the method's argument `r` instead of `radius`. However, `radius` is certainly 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`" refers 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 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
   T getXxx() {
      return xxx;
   }

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

For a boolean variable xxx, the getter shall be named `isXxx()`, instead of `getXxx()`, as follows:

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

// Getter
public boolean isXxx() {
   return xxx;
}

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

Notes:

- `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 the object. You can invoke the `toString()` method explicitly by calling *anInstanceName*`.toString()` or implicitly via `println()` or `String` concatenation operator '`+`'. Running `println(`*anInstance*`)` with an object argument invokes the `toString()` method of that instance implicitly.

For example, if the following `toString()` method is included in our `Circle` class:

```
// Return a short String description of this instance
public String toString() {
   return "Circle with radius = " + radius + " and color of " + color;
}
```

In your `TestCircle` class, you can get a short text descriptor of a `Circle` object by:

```
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 c1.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 address.
3. A `final` class cannot be sub-classed (or extended).
4. A `final` method cannot be overridden.

2.17 Putting Them Together: Circle Class in the Revised

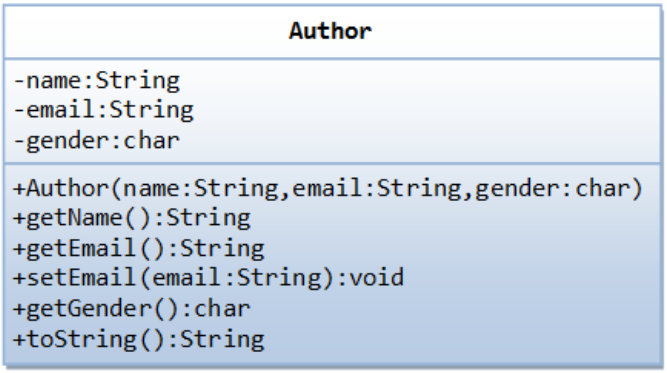
```
1  // The Circle class definition
2  public class Circle {    // Save as "Circle.java"
3     // Public constants
4     public static final double DEFAULT_RADIUS = 8.8;
```

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

   // Private variables
   private double radius;
   private String color;

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

   // Public getter and setter for private variables
   public double getRadius() {
      return radius;
   }
   public void setRadius(double radius) {
      this.radius = radius;
   }
   public String getColor() {
      return color;
   }
   public void setColor(String color) {
      this.color = color;
   }

   // toString() to provide a short description of this instance
   public String toString() {
      return "Circle with radius = " + radius + " and color of " + color;
   }

   // Public methods
   public double getArea() {
      return radius*radius*Math.PI;
   }
}
```

3. More Examples on Classes

3.1 A Tutorial and Book on Classes

A class called `Author` is defined as shown in the class diagram. It contains:



- Three `private` member variables: `name` (`String`), `email` (`String`), and `gender` (`char` of either `'m'`, `'f'`, or `'u'` for unknown - you might also use a `boolean` variable called `male` having value of `true` or `false`).
- One constructor to initialize the `name`, `email` and `gender` with the given values.
  (There is no *default constructor* for `Author`, as there are no defaults for `name`, `email` and `gender`.)
- Public getters/setters: `getName()`, `getEmail()`, `setEmail()`, and `getGender()`.
  (There are no setters for `name` and `gender`, as these attributes cannot be changed.)
- A `toString()` method that returns "`author-name (gender) at email`", e.g., "`Tan Ah Teck (m) at ahTeck@somewhere.com`".

Author.java

```
// The Author class definition
public class Author {
   // Private variables
   private String name;
   private String email;
   private char gender;
```

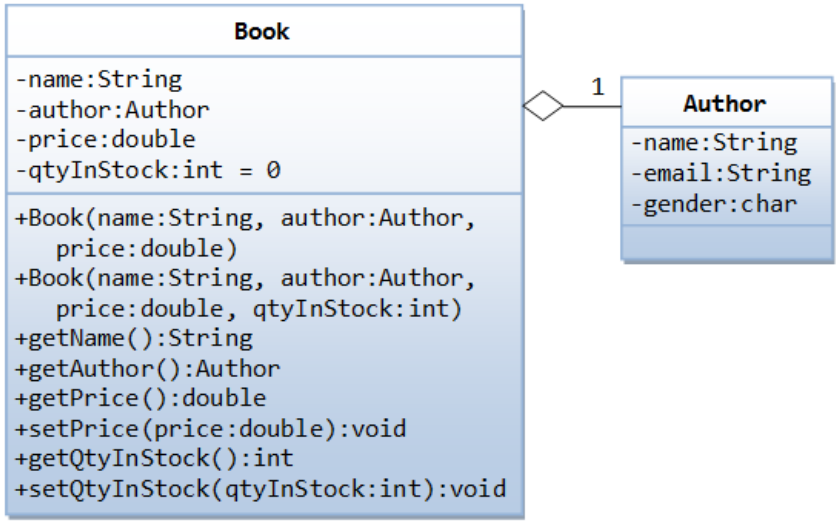
```
7
8     // Constructor
9     public Author(String name, String email, char gender) {
10       this.name = name;
11       this.email = email;
12       this.gender = gender;
13    }
14
15    // Public getters and setters for private variables
16    public String getName() {
17       return name;
18    }
19
20    public String getEmail() {
21       return email;
22    }
23
24    public char getGender() {
25       return gender;
26    }
27
28    public void setEmail(String email) {
29       this.email = email;
30    }
31
32    // toString() to describe itself
33    public String toString() {
34       return name + " (" + gender + ") at " + email;
35    }
36 }
```

A Test Driver: TestAuthor.java

```
1  // A test driver for the Author class
2  public class TestAuthor {
3     public static void main(String[] args) {
4        Author teck = new Author("Tan Ah Teck", "teck@somewhere.com", 'm');
5        System.out.println(teck);  // toString()
6        teck.setEmail("teck@nowhere.com");
7        System.out.println(teck);  // toString()
8     }
9  }
```

A class called Book is defined as shown in the class diagram. It contains:

- Four `private` member variables: `name` (`String`), `author` (an *instance* of the class `Author` you have just created, assume that each book has one and only one author), `price` (`double`), and `qtyInStock` (`int`).
- Two constructors.
- Getters/Setters: `getName()`, `getAuthor()`, `getPrice()`, `setPrice()`, `getQtyInStock()`, `setQtyInStock()`.
- A `toString()` that returns "'book-name' by author-name (gender) at email".
  You should reuse the `Author`'s `toString()` method, which returns "`author-name (gender) at email`".




```
1  // The Book class definition
2  public class Book {
3     // Private variables
4     private String name;
5     private Author author;
6     private double price;
7     private int qtyInStock;
8
9     // Constructors
10    public Book(String name, Author author, double price) {
11       this.name = name;
12       this.author = author;
13       this.price = price;
14       this.qtyInStock = 0;   // Not given, set to the default value
15    }
```

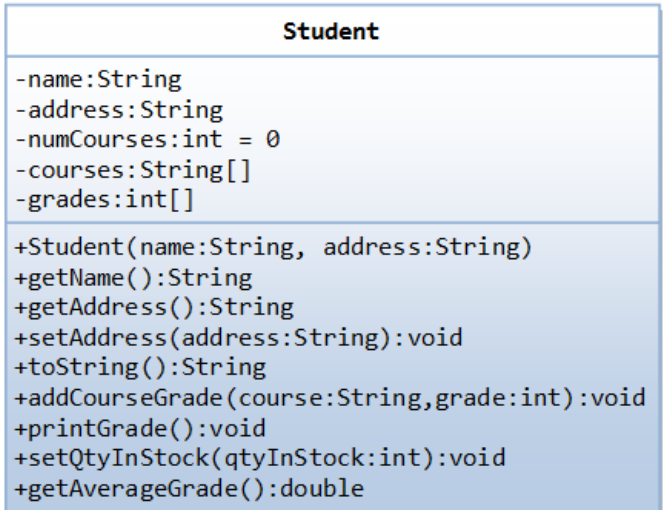
```
16
17      public Book(String name, Author author, double price, int qtyInStock) {
18         this.name = name;
19         this.author = author;
20         this.price = price;
21         this.qtyInStock = qtyInStock;
22      }
23
24      // Getters and Setters
25      public String getName() {
26         return name;
27      }
28
29      public Author getAuthor() {
30         return author;   // return member author, which is an instance of class Author
31      }
32
33      public double getPrice() {
34         return price;
35      }
36
37      public void setPrice(double price) {
38         this.price = price;
39      }
40
41      public int getQtyInStock() {
42         return qtyInStock;
43      }
44
45      public void setQtyInStock(int qtyInStock) {
46         this.qtyInStock = qtyInStock;
47      }
48
49      // toString() t describe itself
50      public String toString() {
51         return "'" + name + "' by " + author;  // author.toString()
52      }
53   }
```

A Test Driver Program: TestBook.java

```
1  // A test driver program for the Book class
2  public class TestBook {
3     public static void main(String[] args) {
4        Author teck = new Author("Tan Ah Teck", "teck@somewhere.com", 'm');
5        System.out.println(teck);  // toString()
6
7        Book dummyBook = new Book("Java for dummies", teck, 9.99, 88);
8        System.out.println(dummyBook);  // toString()
9
10       Book moreDummyBook = new Book("Java for more dummies",
11             new Author("Peter Lee", "peter@nowhere.com", 'm'),  // anonymous instance of Author
12             19.99, 8);
13       System.out.println(moreDummyBook);  // toString()
14    }
15 }
```

3.2 Student Class

Suppose that our application requires us to model a group of students. A student has a name and 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. The class contains:

- Private member variables `name` (`String`), `address` (`String`), `numCourses`, `course` and `grades`. The `numCourses` maintains the number of courses taken by the 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 shall not 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(address) course1:grade1, course2:grade2,...`". You should reuse the `toString()` here.
- A method `getAverageGrade()`, which returns the average grade of all the courses taken.

UML Notations: In UML notations, a variable is written as `[+|-]`*`varName:type`*; a method is written as `[+|-]`*`methodName(arg1:type1, arg2:type2,...):returnType`*, where '`+`' denotes `public` and '`-`' denotes `private`.

The source code for `Student.java` is as follows:

```
1  // The student class definition
2  public class Student {
3     // Private member variables
4     private String name;
5     private String address;
6     // Courses taken and grades for the courses are kept in 2 arrays of the same length
7     private String[] courses;
8     private int[] grades;     // A grade is between 0 to 100
9     private int numCourses;   // Number of courses taken so far
10    private static final int MAX_COURSES = 30;  // maximum courses
11
12    // Constructor
13    public Student(String name, String address) {
14       this.name = name;
15       this.address = address;
16       courses = new String[MAX_COURSES];  // allocate arrays
17       grades = new int[MAX_COURSES];
18       numCourses = 0;                     // no courses so far
19    }
20
21    // Public getter for private variable name
22    public String getName() {
23       return name;
24    }
25
26    // Public getter for private variable address
27    public String getAddress() {
28       return address;
29    }
30
31    // Public setter for private variable address
32    public void setAddress(String address) {
33       this.address = address;
34    }
35
36    // Describe itself
37    public String toString() {
38       return name + "(" + address + ")";
39    }
40
41    // Add a course and grade
42    public void addCourseGrade(String course, int grade) {
43       courses[numCourses] = course;
44       grades[numCourses] = grade;
45       numCourses++;
46    }
47
48    // Print all courses taken and their grades
49    public void printGrades() {
50       System.out.print(this);   // toString()
51       for (int i = 0; i < numCourses; i++) {
52          System.out.print(" " + courses[i] + ":" + grades[i]);
53       }
54       System.out.println();
55    }
56
57    // Compute the average grade
58    public double getAverageGrade() {
59       int sum = 0;
60       for (int i = 0; i < numCourses; i++ ) {
61          sum += grades[i];
```

```
      }
      return (double)sum/numCourses;
   }
}
```

Let us write a test program to create a student named "Tan Ah Teck", who has taken 3 courses, "IM101", "IM102" and "IM103" with grades of 89, 57, and 96 respectively. We shall print all the course grades, and the average grade.

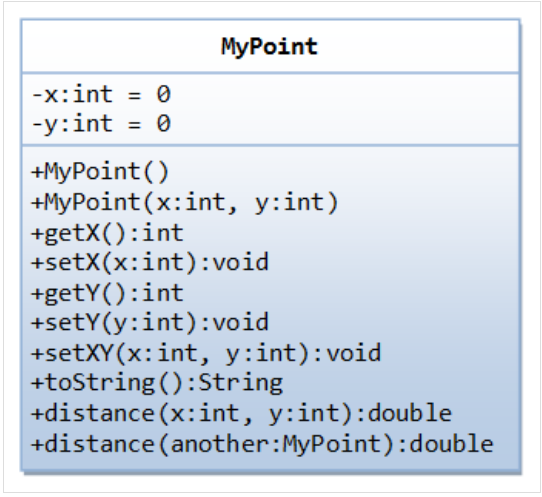
```
// A test driver program for the Student class
public class TestStudent {
   public static void main(String[] args) {
      Student ahTeck = new Student("Tan Ah Teck", "1 Happy Ave");
      ahTeck.addCourseGrade("IM101", 89);
      ahTeck.addCourseGrade("IM102", 57);
      ahTeck.addCourseGrade("IM103", 96);
      ahTeck.printGrades();
      System.out.printf("The average grade is %.2f", ahTeck.getAverageGrade());
   }
}
```

```
Tan Ah Teck(1 Happy Ave) IM101:89 IM102:57 IM103:96
The average grade is 80.67
```

3.3 MyPoint and MyCircle Classes

A class called `MyPoint`, which models a 2D point with x and y coordinates, is designed as shown in the class diagram. It contains:

- Two private member variables x (`int`) and y (`int`).
- A *default* (*no-arg*) constructor that construct a point at `(0,0)`.
- A constructor that constructs a point with the given x and y coordinates.
- Getter and setter for the private variables x and y.
- A method `setXY()` to set both x and y.
- A `toString()` method that returns a string description of the instance in the format "`(x,y)`".
- A method called `distance(int x, int y)` that returns the distance from this point to another point at the given `(x,y)` coordinates in `double`.
- An overloaded `distance(MyPoint another)` method that returns the distance from this point to the given `MyPoint` instance called `another`.



MyPoint.java

```
// The MyPoint class definition
public class MyPoint {
   // Private member variables
   private int x;
   private int y;

   // Constructors
   public MyPoint() {
      x = 0;
      y = 0;
   }

   public MyPoint(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // Getters and Setters
   public int getX() {
      return x;
   }

   public int getY() {
      return y;
   }

   public void setX(int x) {
      this.x = x;
   }

   public void setY(int y) {
      this.y = y;
```

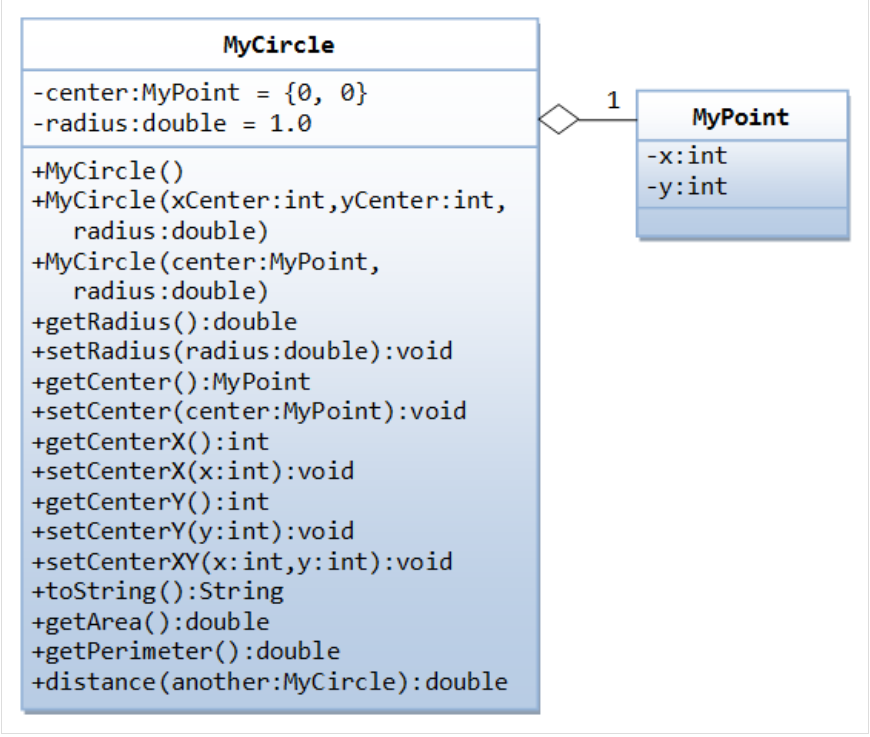
```
   }

   public void setXY(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public String toString() {
      return "(" + x + "," + y + ")";
   }

   public double distance(int x, int y) {
      int xDiff = this.x - x;
      int yDiff = this.y - y;
      return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
   }

   public double distance(MyPoint another) {
      int xDiff = this.x - another.x;
      int yDiff = this.y - another.y;
      return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
   }
}
```

TestDriver for MyPoint.java

```
// A test driver for MyPoint class
public class TestMyPoint {
   public static void main(String[] args) {
      MyPoint p1 = new MyPoint();
      System.out.println(p1);
      p1.setXY(0, 3);
      System.out.println(p1);

      MyPoint p2 = new MyPoint(4, 0);
      System.out.println(p2);

      // Test distance() methods
      System.out.println(p1.distance(4, 0));
      System.out.println(p1.distance(p2));
   }
}
```

A class called `MyCircle` is designed as shown in the class diagram. It contains:

- Two `private` member variables: a `radius` (double) and a `center` (an instance of `MyPoint`, which we created earlier).
- A *default* (*no-arg*) constructor that construct a `Circle` at `(0,0)` with `radius` of 1.0.
- A constructor that constructs a `Circle` with the given `xCenter`, `yCenter` and `radius`.
- A constructor that constructs a `Circle` with the given instance of `MyPoint` as center; and `radius`.
- Getter and setter for the private variables `center` and `radius`.
- Methods `getCenterX()`, `setCenterX()`, `setCenterXY()`, etc.
- A `toString()` method that returns a string description of the instance in the format "`center=(x,y) radius=r`".
- A `distance(MyCircle another)` method that returns the distance from this `Circle` to the given `MyCircle` instance called `another`.



MyCircle.java

```
// The MyCircle class definition
public class MyCircle {
   // Private member variables
```

```
   private MyPoint center;    // Declare instance center
   private double radius;

   // Constructors
   public MyCircle() {
      center = new MyPoint(); // Construct instance at (0,0)
      radius = 1.0;
   }

   public MyCircle(int xCenter, int yCenter, double radius) {
      center = new MyPoint(xCenter, yCenter); // Construct instance
      this.radius = radius;
   }

   public MyCircle(MyPoint center, double radius) {
      this.center = center;
      this.radius = radius;
   }

   // Getters and Setters
   public double getRadius() {
      return radius;
   }

   public void setRadius(double radius) {
      this.radius = radius;
   }

   public MyPoint getCenter() {
      return center;
   }

   public void setCenter(MyPoint center) {
      this.center = center;
   }

   public int getCenterX() {
      return center.getX();
   }

   public void setCenterX(int x) {
      center.setX(x);
   }

   public int getCenterY() {
      return center.getY();
   }

   public void setCenterY(int y) {
      center.setY(y);
   }

   public void setCenterXY(int x, int y) {
      center.setX(x);
      center.setY(y);
   }

   public String toString() {
      return "center=" + center + " radius=" + radius;
   }

   public double getArea() {
      return Math.PI * radius * radius;
   }

   public double getPerimeter() {
      return 2.0 * Math.PI * radius;
   }

   public double distance(MyCircle another) {
      return center.distance(another.center);  // use distance() of MyPoint
   }
}
```

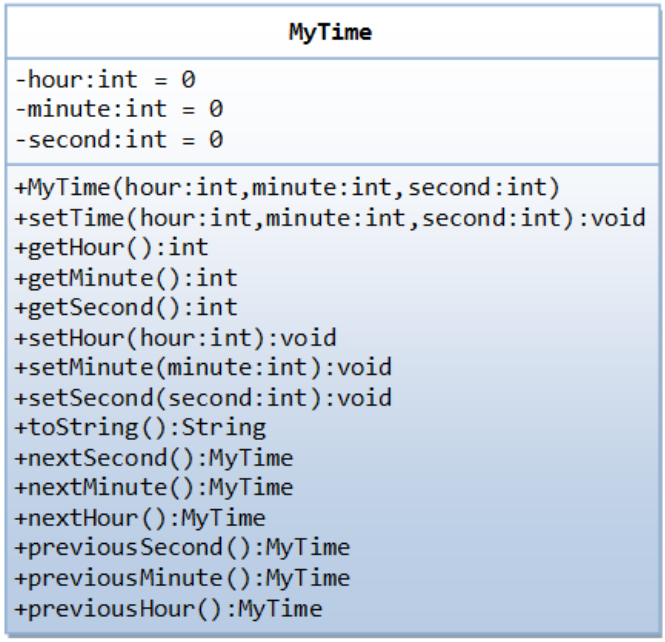
TestMyCircle.java

```
// A test driver for MyCircle class
public class TestMyCircle {
   public static void main(String[] args) {
      MyCircle c1 = new MyCircle();
```

```
      System.out.println(c1);
      c1.setCenterXY(0, 3);
      System.out.println(c1);

      MyPoint p1 = new MyPoint(4, 0);
      MyCircle c2 = new MyCircle(p1, 9);
      System.out.println(c2);

      // Test distance() methods
      System.out.println(c1.distance(c2));
   }
}
```

3.4 MyTimeClass

A class called `MyTime`, which models a time instance, is designed as shown in the class diagram.



It contains the following private instance variables:

- hour: between `00` to 23.
- `minute`: between `00` to 59.
- Second: between `00` to 59.

The constructor shall invoke the `setTime()` method (to be described later) to set the instance variable.

It contains the following public methods:

- `setTime(int hour, int minute, int second)`: It shall check if the given hour, `minute` and `second` are valid before setting the instance variables.
  (Advanced: Otherwise, it shall throw an `IllegalArgumentException` with the message "`Invalid hour, minute, or second!`".)
- `Setters setHour(int hour), setMinute(int minute), setSecond(int second)`: It shall check if the parameters are valid, similar to the above.
- Getters `getHour()`, `getMinute()`, `getSecond()`.
- `toString()`: returns "`HH:MM:SS`".
- `nextSecond()`: Update this instance to the next second and return this instance. Take note that the `nextSecond()` of `23:59:59` is `00:00:00`.
- `nextMinute()`, `nextHour()`, `previousSecond()`, `previousMinute()`, `previousHour()`: similar to the above.

MyTime.java

```
// The MyTime class definition
public class MyTime {    // "MyTime.java"
   // Private member variables
   private int hour;     // 0-23
   private int minute;   // 0-59
   private int second;   // 0-59

   // Constructor
   public MyTime(int hour, int minute, int second) {
      setTime(hour, minute, second);
   }

   void setTime(int hour, int minute, int second) {
      setHour(hour);
      setMinute(minute);
      setSecond(second);
   }

   // Setters which validates input with exception handling
   void setHour(int hour) {
      if (hour >= 0 && hour <= 23) {
         this.hour = hour;
      } else {
         throw new IllegalArgumentException("Invalid hour!");
      }
   }

   void setMinute(int minute) {
      if (minute >= 0 && minute <= 59) {
```

```
30            this.minute = minute;
31         } else {
32            throw new IllegalArgumentException("Invalid minute!");
33         }
34      }
35
36      void setSecond(int second) {
37         if (second >= 0 && second <= 59) {
38            this.second = second;
39         } else {
40            throw new IllegalArgumentException("Invalid second!");
41         }
42      }
43
44      // Getters
45      public int getHour() {
46         return hour;
47      }
48
49      public int getMinute() {
50         return minute;
51      }
52
53      public int getSecond() {
54         return second;
55      }
56
57      // Return description in the format "hh:mm:ss" with leading zeros
58      public String toString() {
59         return String.format("%02d:%02d:%02d", hour, minute, second);
60      }
61
62      // Increment this instance to the next second, return this instance
63      public MyTime nextSecond() {
64         second++;
65         if (second == 60) {
66            second = 0;
67            minute++;
68         }
69         if (minute == 60) {
70            minute = 0;
71            hour++;
72         }
73         if (hour == 24) {
74            hour = 0;
75         }
76         return this;   // Return this instance, to support cascaded operation
77      }
78   }
```

Exception Handling

What to do if an invalid hour, minute or second was given in the constructor or setter? Print an error message? Abruptly terminate the program? Continue operation by setting the parameter to its default? This is a hard decision.

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. For example,

```
void setHour(int hour) {
   if (hour >= 0 && hour <= 23) {
      this.hour = hour;
   } else {
      throw new IllegalArgumentException("Invalid hour!");
   }
}
```

The caller can use the `try-catch` construct to handle the exception (in the test driver below). For example,

```
try {
   MyTime t3 = new MyTime(12, 69, 69);
       // skip remaining statements in try, goto catch
   System.out.println(t1);
} catch (IllegalArgumentException ex) {
   ex.printStackTrace();
}  // after catch, continue next statement
```

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

A Test Driver for the MyTime Class: TestMyTime.java

```
// A test driver program for MyTime
public class TestMyTime {
   public static void main(String[] args) {
      MyTime t1 = new MyTime(23, 59, 58);
      System.out.println(t1);
      System.out.println(t1.nextSecond());
      System.out.println(t1.nextSecond().nextSecond().nextSecond());

      // MyTime t2 = new MyTime(12, 69, 69); // abrupt termination
                                             // NOT continue to next statement

      // Handling exception gracefully
      try {
         MyTime t3 = new MyTime(12, 69, 69);
             // skip remaining statements in try, goto catch
         System.out.println(t1);
      } catch (IllegalArgumentException ex) {
         ex.printStackTrace();
      }  // after try or catch, continue next statement

      System.out.println("Continue after exception!");
   }
}
```

Without the proper try-catch, the "`MyTime t2`" will abruptly terminate the program (i.e., the rest of the program will not be run). With proper try-catch handling, the program can continue its operation (i.e., graceful handling of exception).

3.5 Exercises on Defining Classes

LINK TO EXERCISES

LINK TO JAVA REFERENCES & RESOURCES

Latest version tested: JDK 1.7.0_17
Last modified: April, 2013