OBJECT ORIENTED PROGRAMMING

Lecture notes

Abstract:
Introduce basic knowledge about Object Oriented Programming concepts.
Some examples and exercises are also introduced to help readers catch-up the lesson.

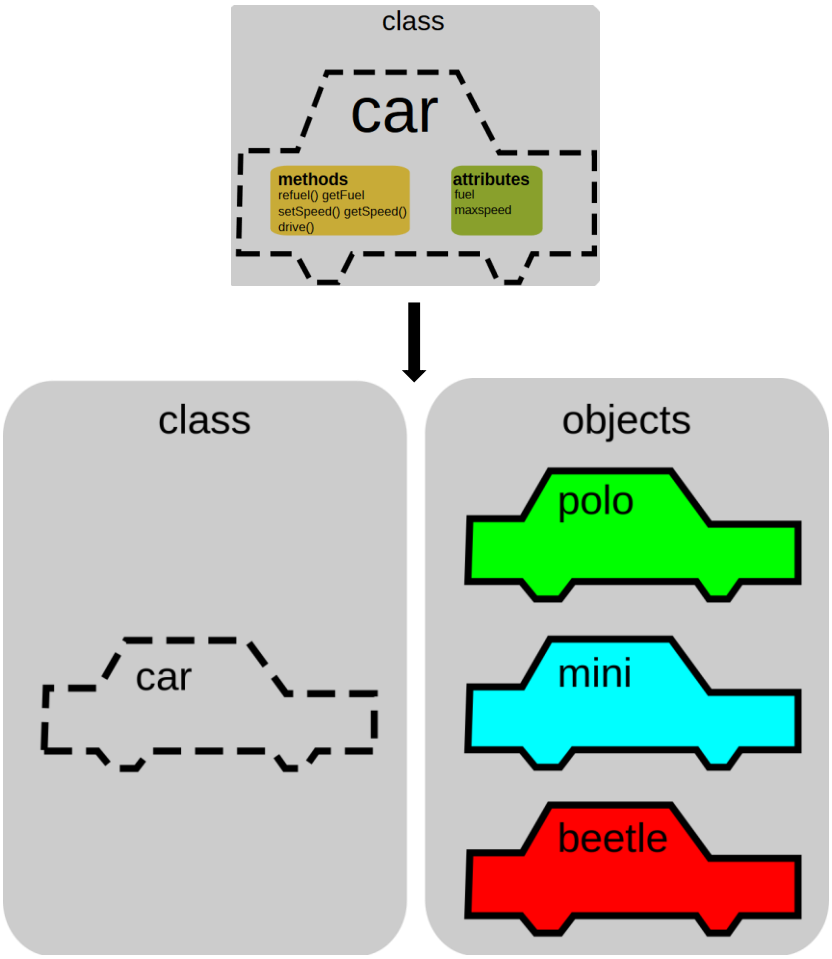
Author:
Tool Development 2 team – Broad Based Tool 2 group

Version 1.1.0

CONTENTS

INTRODUCTION

Object Oriented Programming (OOP) is the term used to describe a programming approach based on objects and classes. The object-oriented paradigm allows us to organize software as a collection of objects that consist of both data and behavior. This is in contrast to conventional functional programming practice that only loosely connects data and behavior.



(Source: https://upload.wikimedia.org/wikipedia/commons/9/98/)

Since the 1980s, the word 'object' has appeared in relation to programming languages, with almost all languages developed since 1990 having object-oriented features. Some languages have even had object-oriented features

retro-fitted. It is widely accepted that object-oriented programming is the most important and powerful way of creating software.

The object-oriented programming approach encourages:

- Modularization: where the application can be decomposed into modules.
- Software re-use: where an application can be composed from existing and new modules.

In this course, we mainly focus on object-oriented mind-set, not coding. PC, laptop will be eliminated from trainees. All things you can do is building up your mind-set about object-oriented. After finishing this course, we hope all trainees can quickly imagine the good design for new software before implementing.
All coding exercises will be done in JAVA, a general-purpose computer-programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. Trainees do not need to learn JAVA to prepare for this course. All related knowledge will be introduced by trainer, that's enough for trainees to finish the exercises.
Note: In this document, you will see some code. But please notice that many of them do not follow any kind of programming language, just a pseudo code to describe the idea.

STATIC VS DYNAMIC

Anything that is created, defined or bound at compile time is *static*. Anything that is created, defined or bound at run time is *dynamic*. Examples:

- If we declare that a variable *i* has type *int*, then that type is statically bound to that variable, but the values assigned at run time to the variable are dynamically bound.
- If a compiler copies all of the object code from library subprograms into the executable file for a program, that is static linking. If the library object code is only accessed at run time, as needed, then that is dynamic linking. JAVA classes are loaded at run time, hence are dynamically linked.
- If a method or variable is defined in a JAVA class as static, then there is just one of them defined, it belongs to the class itself, and is available to use before any dynamic objects are created. If a method or variable is not static, dynamic objects have to be created before they can be used, as many copies are created as objects, and they belong to the objects.

What we need to remember?

.....

.....

.....

.....

.....

.....

CLASSES AND PACKAGES

Nearly all JAVA code is encapsulated in a class which:

- can act as a simple container for related static subprograms and constants, e.g. java.lang.Math;
- can act as an abstract data object, maintaining a single static collection of data (state), and providing the services (operations) that can be performed with that state;
- can act as an abstract data type, providing a template for dynamic objects each of which maintains its own state and the operations that can be performed with it.

A class may mix and match these above patterns, but good designs are usually just one of them.

Example for class: If we think of a real-world object, such as a television, it will have several features and properties:

- We do not have to open the case to use it.
- We have some controls to use it (buttons on the box, or a remote control).

What we need to remember?

.....

.....

.....

.....

.....

.....

- We can still understand the concept of a television, even if it is connected to a DVD player.
- It is complete when we purchase it, with any external requirements well documented.
- The TV will not crash!

A class should:

- Provide a well-defined interface - such as the remote control of the television.
- Represent a clear concept - such as the concept of a television.
- Be complete and well-documented - the television should have a plug and should have a manual that documents all features.
- The code should be robust - it should not crash, like the television.

What we need to remember?

.....

.....

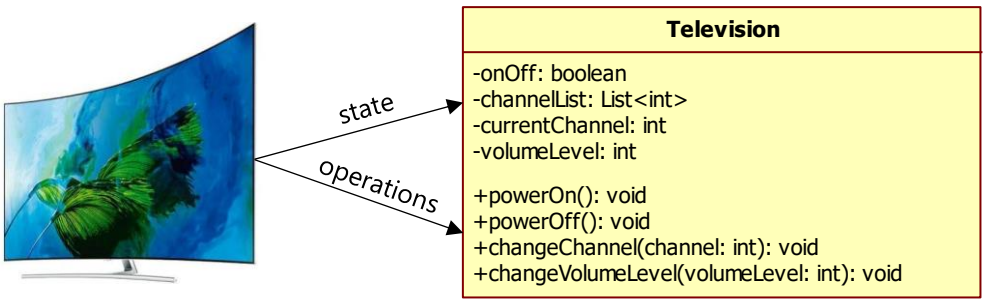
.....

.....

.....

.....

To describe the class visually while designing, we can use class diagram. For example: with Television, we have the state and operation described as below



In JAVA, to define a new class, we use the following syntax:

```
<scope> class <ClassName> {
        // Define something here
}
```

The key <scope> will be introduced clearly in chapter **Member Accessor**. Notice that, we can define another class inside a class. But a careful consideration is needed because there are some limitation with this kind of definition.

What we need to remember?

.....

.....

.....

.....

.....

.....

For example: to define the class Television, we use the following source code:

```
public class Television {
        // Define state in the top of class (Coding convention)
        private boolean onOff;
        private List<int> channelList;
        private int currentChannel;
        private int volumeChannel;

        // Define operation
        private void powerOn() {
                // do something here
        }
        private void powerOff() {
                // do something here
        }
        private void changeChannel(int channel) {
                // do something here
        }
        private void changeVolumeLevel(int volumeLevel) {
                // do something here
        }
}
```

To use class, we define an instance of class:

```
<ClassName> <variableName> = new <ClassName>(<parameter>)
```

What we need to remember?

.....

.....

.....

.....

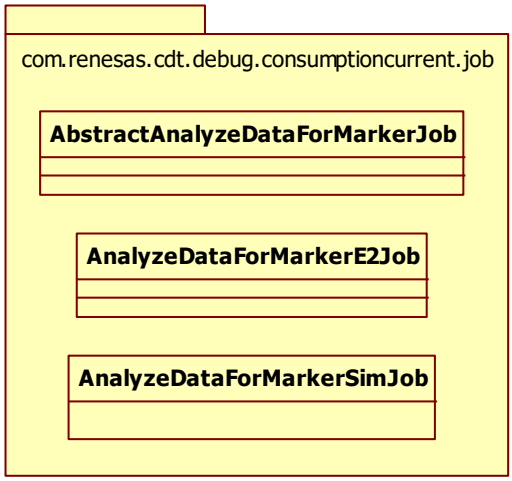
.....

.....

For example, to declare new television object, we will do:

```
Television tvObject = new Television();
```

Packages are collections of classes. Usually, we put some classes have the same purpose into a same package. To describe the package visually, we can use class diagram. For example: while supporting analyze data feature for Measuring Current Consumption plugin for e^2 studio we defined some new jobs, they will be put in the same package



What we need to remember?

.....

.....

.....

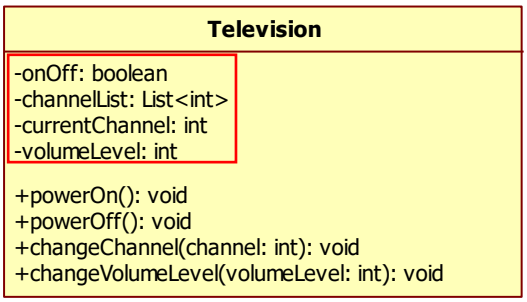
.....

.....

.....

FIELDS, PROPERTIES AND ATTRIBUTES

The global (declared outside of any method) variables of a class are its fields, properties, or attributes. All of these are synonyms. They make up the state of the class (if static) or objects (if dynamic).



Notice that, in front of the fields is "+" or "-" character. They indicate the scope of that field. They will be described clearly in **Member Accessor**.

What we need to remember?

.....

.....

.....

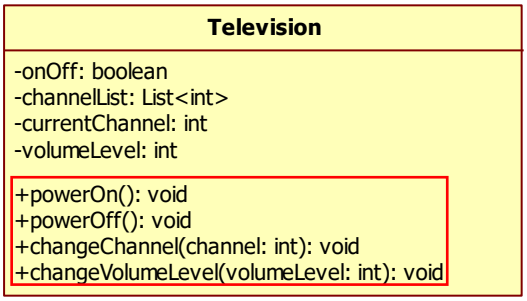
.....

.....

.....

METHODS

The functions and procedures in a class are its methods. They implement the services that the classes or objects provide.



Notice that, in front of the methods is "+" or "-" character. They indicate the scope of that method. They will be described clearly in **Member Accessor**.

What we need to remember?

.....

.....

.....

.....

.....

.....

CONSTRUCTORS AND DESTRUCTORS

The special method that are called when an object is created, are called *constructor*. Their main purpose is to initialize the state of new dynamic objects. Notice that, the constructor is not compulsory to be defined in a JAVA class. In that case, the default constructor will be called by JVM.

For example: we define a class Addition to plus 2 integer numbers

```
public class Addition {
        private int a;
        private int b;

        public void setA(int a) { this.a = a; }
        public void setB(int b) { this.b = b; }
        public int getResult(){ return a+b; }
}
```

To plus 2 integer numbers, we create new instance and set value for 2 integer numbers, then call API to get the result, such as:

```
Addition obj = new Addition(); // Using default constructor
obj.setA(2);
obj.setB(3);
System.out.Println("Result is " + obj.getResult()); // Console displays "Result is 5"
```

What we need to remember?

.....

.....

.....

.....

.....

.....

In the above example, we can reduce the step setting value by define a constructor inside class Addition, such as:

```
public class Addition {
        private int a;
        private int b;

        public Addition(int a, int b) { this.a = a; this.b = b; }
        public void setA(int a) { this.a = a; }
        public void setB(int b) { this.b = b; }
        public int getResult(){ return a+b; }
}
```

Now, to plus 2 integer number we just create new instance with input value then call API to get the result, such as:

```
Addition obj = new Addition(2, 3); // Create new object with a = 2 and b = 3
System.out.println("Result is " + obj.getResult()); // Console displays "Result is 5"
```

What we need to remember?

.....

.....

.....

.....

.....

.....

Of course, reducing LoC is not an only reason to have constructor inside our class. Let's consider the below example: We want to create a class to manage username and password.

```
public class UserManagement {
        private Map<String, String> userInfoMap;

        public boolean isValidUser(String userName, String password);
        public void removeUser(String userName);

        // This API to load all user info from database and save to our map userInfoMap
        public void loadUserInfoFromDatabase(String dbLink);
}
```

Notice that we have API loadUserInfoFromDatabase() to load all user info from system. This API must be called at least one time, if not, there is no data will be stored for using in run-time. So where should we call this API? It may be like that:

```
UserManagement obj = new UserManagement();
obj.loadUserInfoFromDatabase("sqlserver://localhost:1433;DatabaseName=MyDatabase");
```

What we need to remember?

.....

.....

.....

.....

.....

.....

Actually, it's OK. But can we sure developer will always remember to call this API?

There is a way to ensure this, it looks like:

```
public class UserManagement {
        private Map<String,String> userInfoMap;
        private static final String DBLink = ("sqlserver://localhost:1433;DatabaseName=MyBD");

        public UserManagement() {
                loadUserInfoFromDatabase(DBLink);
        }
        public boolean isValidUser(String userName, String password);
        public void removeUser(String userName);

        // This API to load all user info from database and save to our map userInfoMap
        private void loadUserInfoFromDatabase(String dbLink);
}
```

Now, when developer creates new instance of this class, the API will be called.

```
UserManagement obj = new UserManagement();
```

Notice that the database location is now hidden. It's better than making it public to outside.

What we need to remember?

.....

.....

.....

.....

.....

.....

In C++, we have already known about destructor, which is the special subprograms that are called to free the memory when an object is deleted. In JAVA, a destructor is un-needed. And we don't need to free unused memory with "delete" or "malloc" likes in C. It will be done automatically by Garbage Collection.

Note: In the above example, you see "this" keyword. That is a keyword in JAVA to point to current object. In that example, the input of method is "a" and "b". You cannot write: "a=a" and "b=b". JAVA does not understand that you set the inputs of method to the field "a" and "b" of object. So writing "this.a = a" means get the value of input then set to its field "a".

What we need to remember?

.....

.....

.....

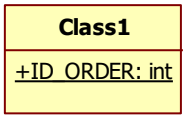
.....

.....

.....

MEMBER ACCESSOR

The members of a class are its fields, methods, constructors, and anything else declared at the top level inside the class. Each member of class can be *dynamic* or *static*. A dynamic member can only be accessed via instance of class. Each dynamic field of each instance are different. A static member can be accessed via both class and instance. Each static field of each instance are same. In class diagram, we describe the static variable as below:



In the above diagram, please notice the sign "+" in front of "ID_ORDER". That sign show the scope of this variable. In next part, that sign will be described clearly.

What we need to remember?

.....

.....

.....

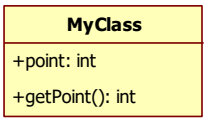
.....

.....

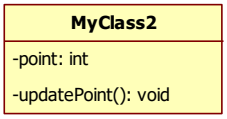
.....

Each member of class (if static) or instance (if dynamic) can have different access level (we can call "scope"). It can be *public*, *private* or *protected*.

- A *public member* can be accessed outside and inside the class. In OOP, only methods are usually public. We use the public method as a provided service. In class diagram, the public member will be described as below:



- A *private member* can only be accessed inside the class. In OOP, fields are usually private. They can be used freely inside the class, by its method. A class may provide some public methods for outside to access to a copy of its field for some reason. In class diagram, we describe private member as below:



What we need to remember?

.....

.....

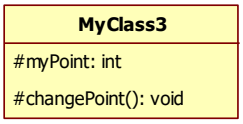
.....

.....

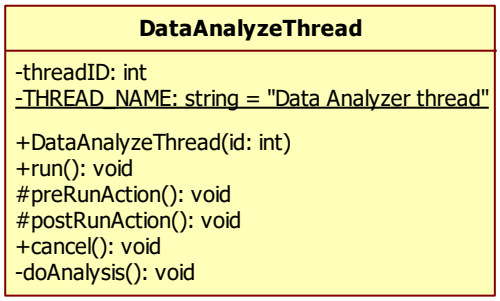
.....

.....

- A *protected member* is a private-public member. It can be accessed by the inheritances of current class or by classes in the same package with current class. (See chapter ***Inheritance***). In class diagram, we describe protected member as below:



See the following class diagram and judge which sentence is correct:



1. "threadID" is static variable.
2. "threadID" is protected variable.
3. "THREAD_NAME" is static variable.
4. "THREAD_NAME" is private variable.
5. "DataAnalyzeThread" is public method.
6. "run" is protected method.
7. "preRunAction" is public method.
8. "postRunAction" is private method.
9. "cancel" is protected method.
10. "doAnalysis" is private method.

What we need to remember?

.....

.....

.....

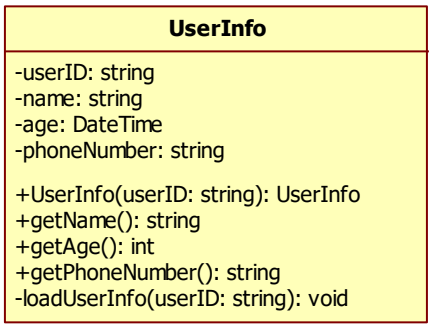
.....

.....

.....

ENCAPSULATION

Encapsulation is the process of combining data and methods into a single unit called class. In Encapsulation, the data is not accessed directly; it is accessed through the methods present inside the class. In simpler words, fields of the class are kept private, then public getter and setter methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible. See the following example:

<table>
<tr>
<td>


</td>
<td>

```
public UserInfo(string userID) {
        this.userID = userID;
        loadUserInfo(userID);
}
private void loadUserInfo(string userID) {
        // connect to database then store data to local
        // variable: name, age, phoneNumber
        ...
}
```

</td>
</tr>
<tr>
<td colspan="2">

```
UserInfo user = new UserInfo();
user.getAge();
```

</td>
</tr>
</table>

What we need to remember?

.....

.....

.....

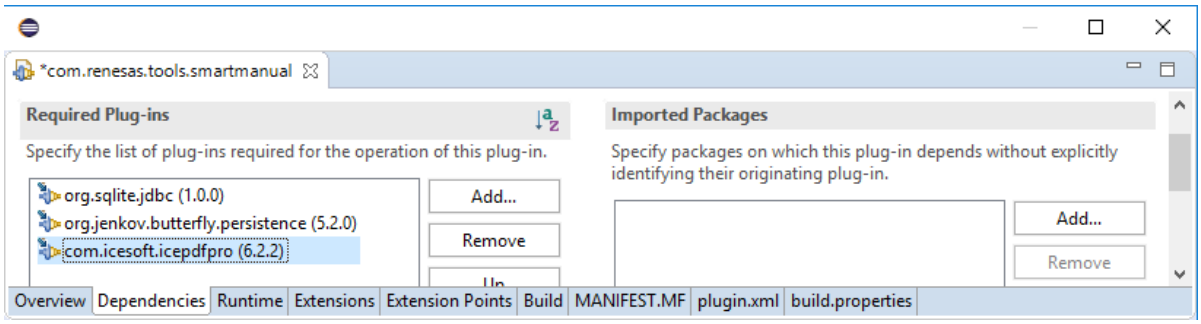
.....

.....

.....

By calling "user.getAge()", we can know the age of user, but we cannot know the day of birth. Class UserInfo has hidden its data. Also, notice on private method "loadUserInfo()", it has been called in Constructor. When creating an instance of UserInfo class, we're unable to know how it can retrieve the information inside. The technology and database had been protected well.

In fact, we usually use API Specification (or we call User Manual) to read the available services provided by classes. We could not know the implement inside. Example: To develop Smart Manual plugin for e^2 studio, Renesas bought ICEpdf library (a PDF reader/viewer plugin on JAVA) from ICEsoft. They send us the load module and User Manual only. We imported the load module and called the provided services to implement our plugin:



What we need to remember?

.....

.....

.....

.....

.....

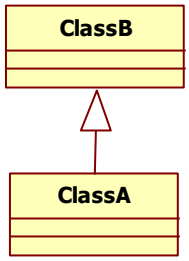
.....

INHERITANCE

Inheritance is the capability of a class to use the fields and methods of another class while adding its own functionality. A sub-class inherits all *public* and *protected* members of its parent. It's a mechanism for sharing/reusing source code. In JAVA, to inherit a class, we use the following syntax:

```
public class ClassA extends ClassB {
        // Something here
}
```

In UML, we describe the inheritance as following:



A class that is derived from another class is called a *subclass* (also a derived class, extended class, or child class). In above example: "ClassA" is subclass.

What we need to remember?

.....

.....

.....

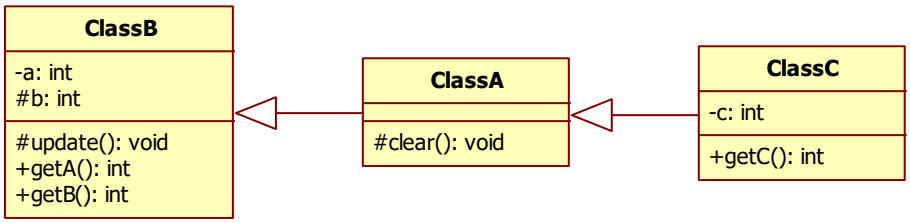
.....

.....

.....

The class from which the subclass is derived is called a *superclass* (also a base class or parent class). In above example: "ClassB" is superclass.

Classes can be derived from classes that are, in turns, derived from other classes. Example:



We create 3 instances of ClassA, ClassB and ClassC as below, which command is correct:

Create 3 instances: ClassA objA = new ClassA(); ClassB objB = new ClassB(); ClassC objC = new ClassC();	1. objA.clear(); 2. objA.getA(); 3. objA.getB(); 4. objA.getC(); 5. objB.getA(); 6. objB.getB(); 7. objB.getC();	8. objB.update(); 9. objC.getA(); 10. objC.getB(); 11. objC.getC(); 12. objC.update(); 13. objC.clear(); 14. objA.update();

What we need to remember?

.....

.....

.....

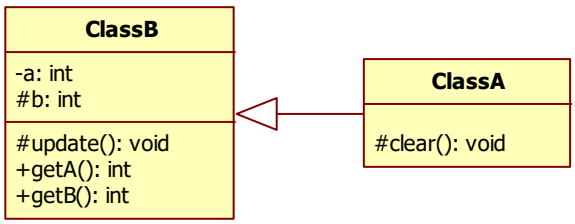
.....

.....

.....

In subclass:

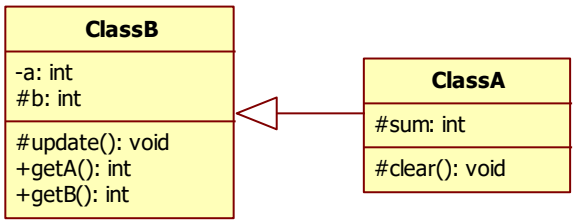
We can declare new methods in the subclass that are not in the superclass. The inherited fields and method from superclass can be used directly.



Inside class ClassA, we can write:

```
protected void clear() {
        b = 0;
        update();
}
```

We can declare new fields in the subclass that are not in the superclass.



Inside class ClassA, we can write:

```
protected void clear() {
        b = 0;
        sum = 0;
}
```

Note: In JAVA, multiple inheritance is not supported, means: ClassA cannot inherit both ClassB and ClassC.

What we need to remember?

.....

.....

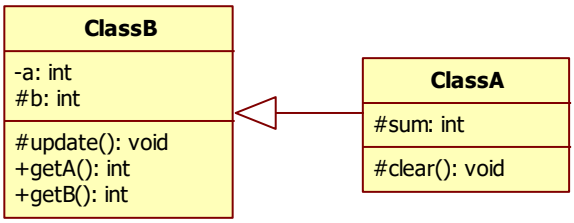
.....

.....

.....

.....

We can override a method or call superclass method.



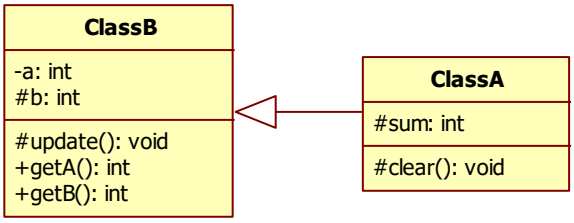
Assume, in class ClassB we have:

```
protected void update() {
        a++;
        b += a;
}
```

Inside class ClassA, we can write:

```
@Override
protected void update() {
        if (b > 100) {
                b++;
        } else {
                super.update();
        }
}
```

We can cast subclass to superclass, but can NOT cast from superclass to subclass.




```
ClassA objA = new ClassA();
ClassB objB = ((ClassB) objA);
```

What we need to remember?

.....

.....

.....

.....

.....

.....

POLYMORPHISM

Polymorphism means many forms. It uses the one method name, but with many different implementations to handle different data types. For example in JAVA.lang.Math we have abs(int), abs(long), abs(double), …

In implementation aspects, there are 2 types of polymorphism.

- Static or compile-time
- Dynamic

Static polymorphism allows you to implement multiple methods within the same class that use the same name but has a different set of parameters (That is called method overloading).

The parameter sets have to differ in at least one of the following three criteria:

- *They need to have a different number of parameters.*

  Example: One method accepts 1 and another one 2 parameters.

  Name(String fullName), Name(String firstName, String lastName)
- *The types of the parameters need to be different.*

  Example: Age(int number), Age(String letters)

What we need to remember?

……

……

……

……

……

……

- *They need to expect the parameters in a different order.*

  Example: Person(String name, int age), Person(int age, String name)

Dynamic Polymorphism cannot be determined by compiler in executed method. It will be determined at runtime by Virtual Machine (e.g. JVM)

As we known, within an inheritance hierarchy, a subclass can override a method of its superclass. This behavior may cause some common mistake. Let's look on following classes definition:

```
Class Father {
    protected String laugh() {
      System.out.println("Haha");
    }
}
```

```
Class Son extends Father {
    protected String laugh() {
      System.out.println("Hehe");
    }
}
```

Now the following command is error at compile time:

```
Son s2 = (Son) new Father(); // Error at runtime
```

What we need to remember?

.....

.....

.....

.....

.....

.....

The following commands show us the difference between Father.laugh() and Son.laugh():

```
Father f1 = new Father();
f1.laugh(); // “Haha”
Son s1 = new Son();
s1.laugh();  // “Hehe”
```

As we known, within an inheritance hierarchy, subclass can be casted to superclass. When combining Inheritance with Polymorphism, we can meet the following common mistake:

```
Son s1 = new Son();
s1.laugh();  // “Hehe”
Father f1 = ((Father) s1); // We cast object here
f1.laugh(); // Do you think it’s “Haha”??? If yes, you’ve made the most common mistake.
```

What we need to remember?

.....

.....

.....

.....

.....

.....

ABSTRACTION

Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. If we define only the prototype of method inside class, that method is an *abstract method*. The class that contains abstract method will be called *abstract class*.

In JAVA, abstract class is a class which contains the **abstract** keyword.

- Abstract classes may or may not contain abstract methods, e.g. methods without body: public void get();
- If a class has at least one abstract method, then the class must be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- To use an abstract class, we have to inherit it from another class, provide implementations to the abstract methods in it. If we inherit an abstract class, we have to provide implementations to all the abstract methods in it.

What we need to remember?

.....

.....

.....

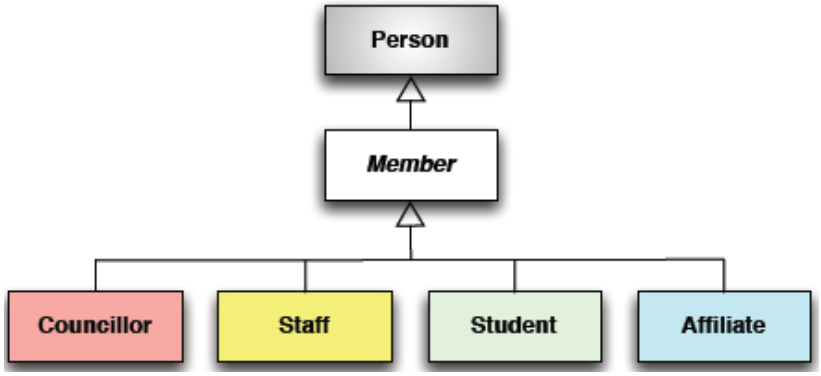
.....

.....

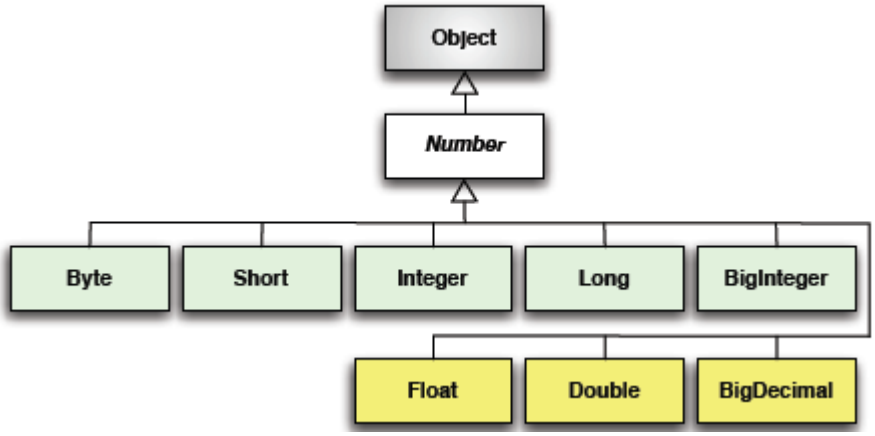
.....

In UML, abstract classes are italicized.

Example 1: Member is an abstract class.



Example 2: All of the wrapper classes for numbers are subclasses of abstract class JAVA.lang.Number, and so are a few more



What we need to remember?

.....

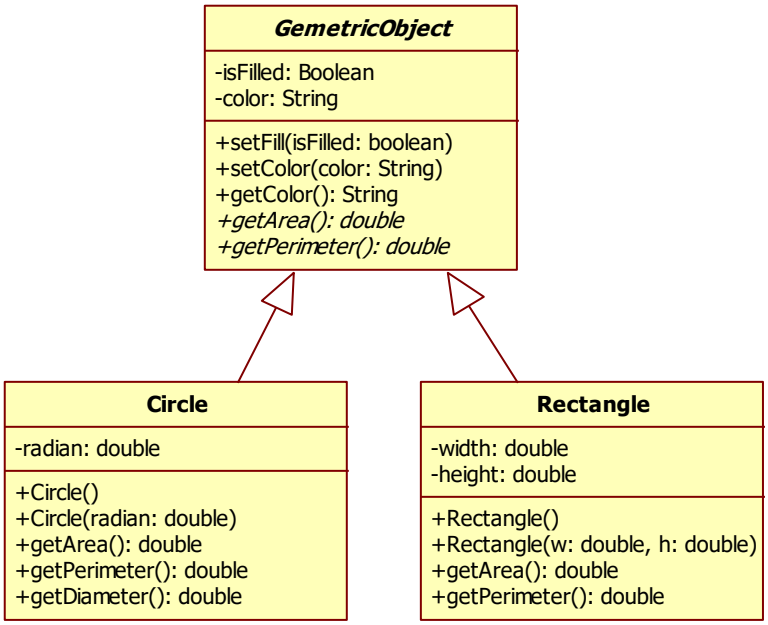
.....

.....

.....

.....

.....




```
public abstract class GeometricObject {
private boolean isFilled;
private String color;

public void setFilled(boolean isFilled) {
        this.isFilled = isFilled;
}

public void setColor(String color) {
        this.color = color;
}

public abstract double getArea();

public abstract double getPerimeter();

}
```

What we need to remember?

.....

.....

.....

.....

.....

.....

Now, we can define new class from an abstract class:

```
public class Circle extends GeometricObject{
private double radian;
public Circle() {
        radian = 5;
}
public Circle (double radian) {
        this.radian = radian;
}

public double getArea() {
        return radian * radian * 3.14;
}

public double getPerimeter() {
        return radian * 2 * 3.14;
}
public double getDiameter() {
        return radian * 2;
}
}
```

```
// Trainee can created Rectangle class by your self
```

Of course, we can do the following with Circle object:

```
Circle = new Circle(10);
circle.setColor("red");
```

What we need to remember?

.....

.....

.....

.....

.....

.....

INTERFACE

Interface is a shared boundary across which two or more separate components of a computer system exchange information.

In JAVA, the **interface** keyword is used to declare an interface with some features:

- Cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Abstract class vs Interface

Abstract class	Interface
Abstract class is a class. It usually defines some default implementations and provides some tools useful for a full implementation.	An interface is a contract. It is used for APIs to exchange information. An interface is an empty shell. It's just a pattern.

What we need to remember?

.....

.....

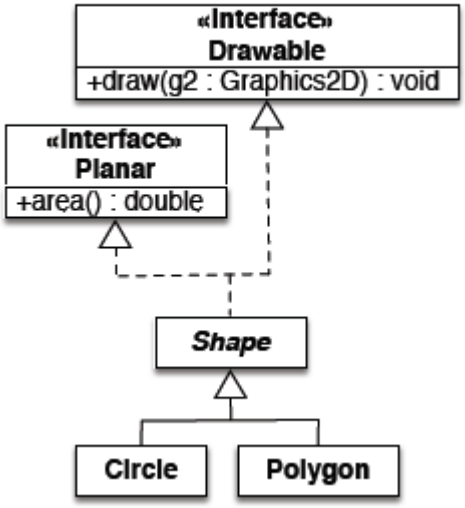
.....

.....

.....

.....

Let's see the following example: Shape class implements Drawable and Planar interface



public interface Planar {
public double area();
}
	public abstract class Shape
	implements Planar, Drawable {
	protected double x, y; // location
	}

What we need to remember?

.....

.....

.....

.....

.....

.....

ABSTRACTION OR INTERFACE?

Both Abstract class and Interface contain abstract method. Why we need Interface? Is only Abstract class enough for software development? In this chapter, we will find the answer for those questions. Let's consider below practices.

Practice 1: As we know, JAVA does not allow multiple inheritance. So that when your class need to re-use the format from many existing class, implementing multiple interface is a solution. For example, your program'd already had the following behaviors:

```
public interface ICompare {
        public boolean isEqual(Object obj);
}

public interface ISorter {
        public enum DIRECTOR{ASC, DES};
        public void sort(DIRECTOR dir);
}
```

In which, "ICompare" provides service to compare 2 objects and "ISorter" provides service to sort a list of object ascending or descending. Now you need

What we need to remember?

.....

.....

.....

.....

.....

.....

to upgrade your program by adding new kind of data that needs both behaviors comparison and sorting. It's very easy to do, like this:

```
public class NewDataTypeClass implements ICompare, ISorter {
        public boolean isEqual(Object obj) {
                // do something here
        }
        public void sort(DIRECTOR dir) {
                // do something here
        }
}
```

Of course, if your program is using abstract class or non-abstract class in this case, you will face with a big problem.

→ Lesson: If your object needs to re-use format from some existing types, using Interface is a good idea.

Practice 2: As we known, modularization is a trend of software development. It's very complicated for user to re-install everything just to get the new feature of your program. In fact, many programs are very flexible for supporting new features (we usually call "plug-in"). User just needs to download the plug-in on

What we need to remember?

.....

.....

.....

.....

.....

.....

the Internet then place in the installation folder. Then they can use a new feature. How can we do the same thing with our software?

For example, your software is a multimedia player and they currently can play mp3, mp4 and wav files. Now you want your software to play an avi file.

Firstly, you need to notice that to support new features, your software needs to be changed. Assume that your software is not using Interface and to play a file, your software needs a switch-case statement, like this:

```
// This method will play a media file
public void playMedia(MediaFile file) {
   switch (file.type())
      case TYPE.MP3: {
         StreamMusicData data = decodeMP3(file);
         playMusic(data);
         break;
      }
      case TYPE.MP4: {
         StreamVideoData data = decodeMP4(file);
         playVideo(data);
         break;
      }
      case TYPE.WAV: {
         StreamMusicData data = decodeWAV(file);
         playMusic(data);
         break;
```

What we need to remember?

.....

.....

.....

.....

.....

.....

```
    }
    default: {
      informUnsupport("This format is not supported.");
      break;
    }
}
```

To support avi file, we can do like this:

```
// This method will play a media file
public void playMedia(MediaFile file) {
  switch (file.type())
    case TYPE.MP3: {
      StreamMusicData data = decodeMP3(file);
      playMusic(data);
      break;
    }
    case TYPE.MP4: {
      StreamVideoData data = decodeMP4(file);
      playVideo(data);
      break;
    }
    case TYPE.WAV: {
      StreamMusicData data = decodeWAV(file);
      playMusic(data);
      break;
    }
    case TYPE.AVI: {
      StreamVideoData data = decodeAVI(file);
      playVideo(data);
```

What we need to remember?

.....

.....

.....

.....

.....

.....

```
            break;
        }
        default: {
            informUnsupport("This format is not supported.");
            break;
        }
}
```

Changing source code means you need to re-build your software and release it to customer. Your customer will replace the old software with a new build. In the future, customer will do it every time you support a new format.

Now we will see the power of Interface.

Firstly, we define a standard for the processors of each file type, like this:

```
public interface IMediaPlayerProcessing {
    // Check whether media file is supported by this processing
    public boolean isSupported(MediaFile file);
    // Play media file. Remember to check support in advanced
    public void play(MediaFile file);
}
```

Secondly, we define processing for each file type, like this:

```
// MP3 player processing
public class Mp3PlayerProcessing implements IMediaPlayerProcessing {
    public boolean isSupported(MediaFile file) {
        // do something here
    }
```

What we need to remember?

.....

.....

.....

.....

.....

.....

```
    public void play(MediaFile file) {
        // do something here
    }
}

// MP4 player processing
public class Mp4PlayerProcessing implements IMediaPlayerProcessing {
    public boolean isSupported(MediaFile file) {
        // do something here
    }
    public void play(MediaFile file) {
        // do something here
    }
}

// WAV player processing
public class WavPlayerProcessing implements IMediaPlayerProcessing {
    public boolean isSupported(MediaFile file) {
        // do something here
    }
    public void play(MediaFile file) {
        // do something here
    }
}
```

Thirdly, we will build those classes and put into folder "plugins".

What we need to remember?

.....

.....

.....

.....

.....

.....

Then we create a component that can load those classes at run-time. This component will read all files in "plugins" folder and load all classes that implement IMediaPlayerProcessing interface. Like this:

```
public class PlayerProcessingManagement {
   // A list that contains all media processing in "plugins" folder
   List<IMediaPlayerProcessing> availablePlayers;

   // Load all processing from "plugins" folder
   public void loadPlayers() {
      // Read "plugins" folder and load all files
      Object[] files = LoadClassFromFolder(".\\plugins\\");
      foreach (Object classObj in files) {
         if (classObj instanceOf IMediaPlayerProcessing) {
            availablePlayers.add((IMediaPlayerProcessing) classObj);
         }
      }
   }

   // Play a media file
   public void play(MediaFile file) {
      boolean isPlayed = false;
      foreach (IMediaPlayerProcessing playerProcessing in availablePlayers) {
         // If file is supported, play it
         if (playerProcessing.isSupported(file) = true) {
            playerProcessing.play(file);
            isPlayed = true;
            break;
```

What we need to remember?

.....

.....

.....

.....

.....

.....

```
        }
      }
      if (isPlayed = false) {
        informUnsupport("This format is not supported.");
      }
    }
}
```

In the above source code, the manager will get all available processing and check whether each of them can play a target media file. If one of them can play, the manager let it do that. If not, manager will inform error to user.

So, to play a media file, we change the method like this:

```
// This method will play a media file
public void playMedia(MediaFile file) {
   PlayerProcessingManagement playerManager = new PlayerProcessingManagement();
   playerManager.loadPlayers();
   playerManager.play(file);
}
```

As you can see, the method does not check the file type any more. The manager will do everything. Now, to support AVI file, how can we do?

Notice that all the above source code (manager, playMedia) have been built in your software already. The thing we need to do is creating new plugin for your software, like this:

What we need to remember?

.....

.....

.....

.....

.....

.....

```
// AVI player processing
public class AviPlayerProcessing implements IMediaPlayerProcessing {
   public boolean isSupported(MediaFile file) {
      // do something here
   }
   public void play(MediaFile file) {
      // do something here
   }
}
```

You just need to build above class (only the class is enough), and put the output in "plugins" folder. The manager will automatically load it for your software.

Somebody may confuse that we can use Abstract class in this case or not. The answer is "Yes". But in this case, we just need a standard format for all player processing and all of them do not need to share any implementation, so Interface is enough.

→ Lesson: If your software usually changes (applying new design, supporting new features, etc.) and your modules do not share any common implementation, using Interface is a good idea.

The below practice will show you the power of Abstract class.

What we need to remember?

.....

.....

.....

.....

.....

.....

Practice 3: As we known, code readability and coding convention are required in software development. They help developers can work together and cooperate to develop a software base on some standards and common styles. For an example, a group of developers cooperate to develop a software that contains some asynchronized jobs. Each running job need to do 3 steps:

- Preparing the environment (Set variable, load setting, checking input, etc.)
- Doing some specified actions (Connect to database, Retrieve bank account, Retrieve bank OTP, etc.)
- Clearing the garbage (Clear cache memory, Close database connection, etc.)

To force every developer to follow the above design, we need to use abstract class, like this:

```
public abstract class AbstractAsyncJob {
   // This method will be called from OS after we schedule it with OS.
   // Using “final” to force child class not to override this method.
   public final void run(ProgressMonitor monitor) {
      preRunActions(monitor);
      runActions(monitor);
      postRunActions(monitor);
   }
```

What we need to remember?

.....

.....

.....

.....

.....

.....

```
    // Doing some actions before running the main activities.
    protected abstract void preRunActions(ProgressMonitor monitor);
    // Doing main activities.
    protected abstract void runActions(ProgressMonitor monitor);
    // Doing some actions after running the main activities.
    protected abstract void postRunActions(ProgressMonitor monitor);
}
```

To create a new job to do a specified action, all developers create new class that extends AbstractAsyncJob. For example, to retrieve OTP, we have:

```
public class OTPRetrieveJob extends AbstractAsyncJob {
    // Doing some actions before running the main activities.
    @Override
    protected abstract void preRunActions(ProgressMonitor monitor) {
        // Some code will be here.
    }
    // Doing main activities.
    @Override
    protected abstract void runActions(ProgressMonitor monitor) {
        // Some code will be here.
    }
    // Doing some actions after running the main activities.
    @Override
    protected abstract void postRunActions(ProgressMonitor monitor) {
        // Some code will be here.
    }
}
```

What we need to remember?

.....

.....

.....

.....

.....

.....

Notice that when developer create a new non-abstract class that extends an abstract class, they MUST implement all abstract methods of base class. By this constrain, all developers (not only old members but also newly joined members) will follow the same design.

Moreover, all created jobs will have the same base class, so that our program can manage them efficiently by using a job manager (like Practice 2, we use a List to store and manage all defined job).

→ Lesson: If your software modules needs to share common implementation, using Abstract class is a good idea.

→ Lesson: In JAVA, "final" keyword force child class not to override the method.

What we need to remember?

.....

.....

.....

.....

.....

.....

REVISION HISTORY

Version	Modification	Date	Author(s)
1.0.0	First creation	Mar 21, 2018	Phuong Ng. Hoang Ng.
1.1.0	Add chapter "Abstraction or Interface?" Update information in "Introduction"	Oct 04, 2018	Phuong Ng.

Note - Rule for increasing revision number:
The revision number has the following form: {Major}.{Minor}.{Micro}

- *{Major}* is an integer number, will be increased when there are a big change in document (Re-construct the outline, etc.). The increase is done when all AIs are done after the review.
- *{Minor}* is an integer number, will be increased when there are a small change in document (add new chapter, correct typo mistake, etc.). The increase is done when all AIs are done after the review.
- *{Micro}* is an integer number, will be increased during the modification of author (means: author can decide to increase the {Micro} number by themselves, when document is still draft). The change doesn't need to be reviewed. When changes (of several micro versions) are accepted after the review, {Micro} number will be reset to 0, and {Minor} or {Major} number will be increased base on the change scope, document will be set as "Released".

BIBLIOGRAPHY

1. *"1005ICT Object Oriented Programming Lecture Notes"*, School of Information and Communication Technology Griffith University , 2015.
2. *"Object Oriented Programming"*, Binnur Kurt.
3. *"Object Oriented Programming Through JAVA"*, P. Radha Krishna, 2007.
4. *"OOP Concepts for Beginners: What Is Polymorphism"*, https://dzone.com/articles/oop-concepts-for-beginners-what-is-polymorphism.
5. *"OOPs in JAVA: Encapsulation, Inheritance, Polymorphism, Abstraction"*, https://beginnersbook.com/2013/03/oops-in-JAVA-encapsulation-inheritance-polymorphism-abstraction/
6. *"Module EE402 - Object-oriented Programming with Embedded Systems 2017/18"*, http://ee402.eeng.dcu.ie/