OO Concepts Review

- Class
- Method
- Inheritance
- Method Overloading and overriding

Classes are written in source file, a source file might contains more than one class. Rules for source file:

- Only **ONE** public class per source file.
- The source file name same as the public class name.
- Source file without any public class can have any name for the class.
- Statements in source file should be in the sequence of *package*, *import*, *class declaration*.
- *Import* and *package* should be applied to all classes in the same source file.

NOTE: Property *get* and *set* as in C# do not exist in Java.

Java Keywords

`abstract`	`continue`	`for`	`new`	`switch`
`assert`	`default`	`if`	`package`	`synchronized`
`boolean`	`do`	`goto`	`private`	`this`
`break`	`double`	`implements`	`protected`	`throw`
`byte`	`else`	`import`	`public`	`throws`
`case`	`enum`	`instanceof`	`return`	`transient`
`catch`	`extends`	`int`	`short`	`try`
`char`	`final`	`interface`	`static`	`void`
`class`	`finally`	`long`	`strictfp`	`volatile`
`const`	`float`	`native`	`super`	`while`

Primitives

byte, short, int, long, float, double, char, Boolean

Hexadecimal integers are written by using the prefixes **0x** or **0X**. Example, 9E is written as 0x9E or 0X9E.

Octal integers are written by prefixing the numbers with 0. Example, 567 is written as 0567.

To assign a value to a long, suffix the number with the letter **L** or **l**.

Long, int, short and byte can be expressed in binary by prefixing the binary with **0B** or **0b**. Example, byte twelve = 0B1100;

Unicode character can be specified by using the escape character \u. Example, '\u2299'.

Array

Declare an array: `int[] intArray;`

Allocate memory for 5 integers: `intArray = new int[5];`

A class usually contains:

- Constructor (optional)
- Fields
- Methods

Class Member Access Modifiers

Access Level	From			
	the same class	**child classes**	**classes in the same package**	**classes in other packages**
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default	Yes	No	Yes	No
private	Yes	No	No	No

Notes:
Members of the **java.lang** package are imported automatially. If you need to use the java.lang.String, you don't have t import the class explicitly.

Static Members
Static members can be called without instantiating the class. Example, **System.out**.

The **this** keyword
Use the **this** keyword from any method or constructor to refer to the current object.

```
public class Box{
  int length;
  int width;
  int height;

  public Box(int length, int width, int height){
     this.length = length;
     this.width = width;
     this.height = height;
  }
}
```

In the above example, `this.length` refer to the class-level `length` field.

Accessibility in Inheritance

Within a subclass, you can access its superclass's public and protected methods and fields, but not the private methods.

If both subclass and superclass are in the same package, you can also access the superclass's default methods and fields.

```
package cp6;
public class Pr{
	public void publicMethod(){
	}
	protected void protectedMethod(){
	}
	void defaultMethod(){
	}
}

class Ch extends Pr{
	public void testMethods(){
		publicMethod();
		protectedMethod();
		defaultMethod();
	}
}
```

However, you can expose superclass's non-public methods through subclass. The following code won't compile.

```
package test;
import cp6.Ch;
public class TestAccess{
	public void main(String[] args){
		Ch c = new Ch();
		c.protectedMethod();
	}
}
```

`TestAccess` is not a subclass of `Pr`, it can't access `Pr`'s protected method through its subclass `Ch`.

1. Another aspect that needs to be checked for in methods with parameters that are reference variables are the class types of these variables. We will briefly revise the principles of inheritance in Java as a preliminary to discussing this issue. You may skip this if you are familiar with this concept.

```
package my.edu.utar;

class Human {
   int myKad = 1;

```

```java
   public void identify() {
      System.out.println ("I am a human");
   }
}

class Doctor extends Human {
   int myKad = 5;
   int doctorID = 500;

   public void identify() {
      System.out.println ("I am a doctor");
   }
}

class Surgeon extends Doctor {
   int myKad = 10;
   int surgeonID = 1000;

   public void identify() {
      System.out.println ("I am a surgeon");
   }
}

public class CastingExample {
   public static void main(String[] args) {

      Human h1 = new Human();
      Doctor d1 = new Doctor();
      Surgeon s1 = new Surgeon();

      Human h2 = new Doctor();
      h2.identify();
      System.out.println (h2.myKad);
//    System.out.println (h2.doctorID);
      Human h3 = new Surgeon();
      h3.identify();
      System.out.println (h3.myKad);
//    System.out.println (h3.surgeonID);
//    d1 = new Human();
//    s1 = new Human();
      System.out.println ("Doctor ID  and myKad : ");
      Doctor d2 = (Doctor) h2;
      System.out.println (d2.doctorID);
      System.out.println (((Doctor) h2).doctorID);
      System.out.println (d2.myKad);
      System.out.println ("Surgeon ID and myKad : ");
      Surgeon s2 = (Surgeon) h3;
      System.out.println (s2.surgeonID);
      System.out.println (((Surgeon) h3).surgeonID);
      System.out.println (s2.myKad);
```

```
      Human h4 = new Human();
//    d2 = (Doctor) h4;
//    s2 = (Surgeon) h4;

      System.out.println ("Calling showMyKadID with a human");
      showMyKadID(new Human());
      System.out.println ("Calling showMyKadID with a doctor");
      showMyKadID(new Doctor());
      System.out.println ("Calling showMyKadID with a surgeon");
      showMyKadID(new Surgeon());
   }

   public static void showMyKadID(Human h) {
      h.identify();
      if (h instanceof Surgeon) {
         Surgeon s = (Surgeon) h;
         System.out.println ("Surgeon ID : " + s.surgeonID);
         System.out.println ("Surgeon myKad : " + s.myKad);
      }
      else if (h instanceof Doctor) {
         Doctor d = (Doctor) h;
         System.out.println ("Doctor ID : " + d.doctorID);
         System.out.println ("Doctor myKad : " + d.myKad);
      }
      else {
         System.out.println ("Human myKad : " + h.myKad);
      }
   }

   // This method explicitly expects that its parameter is either
   from
   // the Doctor or Surgeon class type otherwise an Exception is
   thrown

   public static void anotherMethod(Human h) {
      if (!(h instanceof Doctor))
         throw new IllegalArgumentException();

      // rest of code
   }
}
```

In the class `CastingExample`, we see that `Human`, `Doctor` and `Surgeon` are involved in an inheritance hierarchy. `Human` is the base/parent/superclass to `Doctor`, which is the related derived/child/subclass in this hierarchy. Similarly, `Doctor` is the superclass to `Surgeon`, which is the related subclass. The method `identify()` that is originally declared in `Human` is overridden in the subclasses `Doctor` and `Surgeon`.

2. We use reference variables from all these 3 class types and instantiate them with objects from their respective classes.

```
Human h1 = new Human();
Doctor d1 = new Doctor();
Surgeon s1 = new Surgeon();
```

Then, we assign a subclass object (`Doctor`) to a superclass reference variable (`Human h2`).

```
Human h2 = new Doctor();
```

When we invoke `identify`() on `h2`, it is the method of the object that is referred to (`Doctor`) that is called. This is known as **polymorphism** or run time binding, where the JVM dynamically determines the correct method to call at run time based on the class of the object being referred to.

3. When we attempt to access the member variable `myKad` through `h2` or `h3`, we will always obtain the value of `myKad` in the `Human` class; regardless of what object `h2` or `h3` is referring to. This is because the value of member variables are determined at compile time, which is statically evaluated by the JVM **on the basis of the class type of the reference variable**. Attempting to access member variables of the subclass (`doctorID` in `Doctor`, and `surgeonID` in `Surgeon`) through the reference variable of the superclass type (`Human h2` or `h3`) results in a compile time error. Uncomment the following lines in the source file to verify that this is the case.

```
//    System.out.println (h2.doctorID);
…
//    System.out.println (h3.surgeonID);
```

4. To summarise, when an attempt is made to access a variable through a reference variable; only the member variables found in the class of that reference variable are accessible. This is static, or compile time, binding.

   When an attempt is made to call a method through a reference variable; the overridden method in the class of the object being referred to is invoked (rather than in the class of the reference variable). This is known as dynamic, or run time, binding.

5. In the lines shown below, we assigned a subclass object to a superclass reference variable.

```
Human h2 = new Doctor();
…
Human h3 = new Surgeon();
```

The converse, which is assigning a superclass object to a subclass reference variable, is not possible and will result in a compile time error. Uncomment the following lines in the source code to verify it.

```
//    d1 = new Human();
//    s1 = new Human();
```

The idea behind this is that the subclass represents a more detailed version of the superclass; the subclass can potentially inherit everything that belongs the superclass as well as having its own unique members and methods that are not accessible to the superclass.

6. Occasionally we may wish to access the unique members of a subclass object that is being referred to by the superclass reference variable. However, any attempt to do this directly, as in the following lines, results in a compile time error.

```
// System.out.println (h2.doctorID);
…
// System.out.println (h3.surgeonID);
```

To accomplish this, first explicitly cast the superclass reference variable to a subclass type. We can first cast and assign the result to a subclass reference variable, then use this reference variable to access the member variable or we can directly cast and access the member variable at the same time. Examples:

```
Doctor d2 = (Doctor) h2;
System.out.println (d2.doctorID);

// directly cast and access
System.out.println (((Doctor) h2).doctorID);
…
Surgeon s2 = (Surgeon) h3;
System.out.println (s2.surgeonID);

// directly cast and access
System.out.println (((Surgeon) h3).surgeonID);
```

7. The casting operations are successful only because `h2` and `h3` are actually referring to `Doctor` and `Surgeon` objects respectively.
However, the JVM does not check to verify this at compile time. It attempts the cast at run time and if an attempt is made to cast an object that is being referred to a class that is not valid, a run time error occurs.
Uncomment the following lines in the source code and run it. Notice that there is **No** compile time error is indicated, but running the application results in a `ClassCastException`.

```
//    d2 = (Doctor) h4;
//    s2 = (Surgeon) h4;
```

8. Often times, a method will have a parameter of a superclass type which can accept arguments of superclass as well as subclass object types. This is easier and more efficient than having a separate parameter for every unique class type.
The method itself will then need to perform a check on the actual class type of the parameter before performing a casting operation on it in order to access the required member variables. This check is performed using the `instanceof` operator as illustrated in the method `showMyKadID` at the bottom of the class.

The evaluation of `if(h instanceof XXX)` will return true if the object class type that is referred to by `h` is the same as `XXX` or lower than `XXX` in the inheritance hierarchy (i.e. a superclass of `XXX`). Thus, if `h` is pointing to a `Surgeon` object, then the statements

`if(h instanceof Human)`, `if(h instanceof Doctor)` and `if(h instanceof Surgeon)` all will evaluate to true.
This means that if you put the `if(h instanceof Doctor)` block in front of the `if(h instanceof Surgeon)` block in the `if-else-if` structure, the method will not work as expected. Try this for yourself and explain why.

9. There may be situations where there is a clear expectation from the method that the parameter it receives is only of certain class types. For example, consider the method `anotherMethod()` in the class.

   This method expects that its parameter must be either a `Surgeon` or a `Doctor`, but not a `Human`. Therefore a parameter of `Human` type would be considered an invalid value, and a check must be made for this and an Exception thrown if it occurs.

   As a simple exercise, try to explain why the statement `if(h instanceof Human)throw new IllegalArgumentException();` will not work, if compared to the current implementation: `if(!(h instanceof Doctor)) throw new IllegalArgumentException();`.

Error Handling – Catch Exceptions

Isolate code that may cause a runtime error using `try` statement, accompanied by the `catch` and `finally` statements. If an error is encountered, Java stops the processing of `try` block and jump to `catch` block to handle the error.

```
package my.edu.utar.learn;
import java.util.Scanner;

public class TryCatch {
     public static void main(String[] args){
          Scanner scanner = new Scanner(System.in);
          System.out.print("Enter a number: ");
          String input = scanner.next();
          try{
               double number = Double.parseDouble(input);
               System.out.println("Result: " + number);
          }catch(NumberFormatException e){
               System.out.println("Invalid input.");
          }
          scanner.close();
     }
}
```

Polymorphism and interfaces

Polymorphism is an important principle in object oriented programming and design that refers to the general concept of a method being able to demonstrate different functionality or behaviour under different circumstances. We will be using this principle in refactoring our application code to perform certain types of tests.

1. In `WithInheritance.java`, we see that `Human`, `Doctor` and `Surgeon` classes are involved in an inheritance hierarchy. `Human` is the base/parent/superclass to `Doctor`, which is the related derived/child/subclass in this hierarchy. Similarly, `Doctor` is the superclass to `Surgeon`, which is the related subclass. The method `identify`() that is originally declared in `Human` is overridden in the subclasses `Doctor` and `Surgeon`.

WithInheritance.java

```
package my.edu.utar;

class Human {
   public void identify() {
      System.out.println ("I am a human");
   }
}

class Doctor extends Human {
   public void identify() {
      System.out.println ("I am a doctor");
   }
}

class Surgeon extends Doctor {
   public void identify() {
      System.out.println ("I am a surgeon");
   }
}

public class WithInheritance {
   public void showYourIdentity(Human[] humans) {
      System.out.println ("\nIdentifying all array elements : ");
      for (int i = 0; i < humans.length; i++) {
         humans[i].identify();
      }
   }

   public static void main(String[] args) {

      Human[] firstHumanArray = {new Human(), new Doctor(), new
   Surgeon(), new Doctor()};
      Human[] secondHumanArray = {new Doctor(), new Surgeon(), new
   Surgeon(), new Human()};

      WithInheritance wi = new WithInheritance();
      wi.showYourIdentity(firstHumanArray);
      wi.showYourIdentity(secondHumanArray);
   }
}
```

2. The method `showYourIdentity()` accepts a parameter that is an array of reference variables from the class `Human`. It then iterates through the array; invoking the `identify`() method on each element of the array.

   We can assign objects of a subclass type to a superclass reference variable. Thus the `humans` array can contain objects from the class `Human` (the superclass), as well as objects from all subclasses descended from `Human` (`Doctor` and `Surgeon`).

3. In the `main()` method, we create two arrays of `Human` reference variables; each containing different objects from the 3 classes of the inheritance hierarchy. We then pass these arrays to the `showYourIdentity()`. As we iterate through these arrays, the appropriate method of the class of the object being referred to by the array elements is invoked.

   Thus, if an array element refers to a `Doctor` object, then the `identify()` method defined in the `Doctor` class is executed. If the array element refers to a `Surgeon` object, then the `identify()` method defined in the `Surgeon` class is executed. Run the class to verify this.

   This is one form of **polymorphism**, where the **JVM determines the method to call dynamically** at run time. The statement `humans[i].identify();` can produce different functionality or behaviour in different situations. The most important point about polymorphism is that there is no need to perform a check for the object type in the array element `humans[i]` before invoking the `identify()` method on it. The implication of this is that polymorphism allows the production of different functionality or behaviour WITHOUT the need to modify or add on extra code to the code calling the method (`humans[i].identify();`).

4. `WithoutInheritance.java` repeats the previous example, but without the use of inheritance. Note that the 3 classes declared at the top of the file: `Student`, `Car` and `Dog` are not involved in any inheritance hierarchy.

   This is because these 3 entities that are being modelled as classes, cannot be conceptualized in a parent-child or superclass-subclass relationship. In the previous program, it might make sense to say that a `Doctor` is a more specialized form of `Human`, and a `Surgeon` is a more specialized form of `Doctor`. However, we are not able to make similar conceptualizations about a student, car or dog. We certainly would not say a student is a more specialized form of a dog, for example! However, we might be able to say that these different entities can all perform a common method (`makeSound()`), with different functionality displayed for each class that models that entity.

5. We can't apply the same approach as in the previous example; which is to create a method that can accept an array of references to objects from all 3 different classes: `Student`, `Car` and `Dog`, and call the `makeSound()` method on them.

   Since none of these classes are related to each other through inheritance, we cannot create an array whose element types are from them. However, the `Object` class is the

root class for the class inheritance hierarchy in the Java API. Therefore if we create an array of `Object` references, the elements of the array can refer to objects of any other class (including the 3 user-defined classes in this example).
However, if the array elements are of the `Object` class type, we cannot directly invoke the `makeSound`() method on them; because the root class does not contain the `makeSound()` method. We have to cast the array element to one of the 3 class types; and in order to do this correctly, we have to determine the class type of the object referred to by a given array element.

6. In the `main()` method, we create two arrays of `Object` reference variables; each containing different objects from the 3 classes and then pass these arrays to the `makeSomeSounds()`. Run this class and check the output in the console.

WithoutInheritance.java

```java
package my.edu.utar;

class Student {
   public void makeSound() {
      System.out.println("I love software engineering !");
     }
}

class Car {
   public void makeSound() {
      System.out.println("Vroom vroom !");
     }
}

class Dog {
   public void makeSound() {
      System.out.println("Bow wow bow wow !");
     }
}

public class WithoutInheritance {

   public void makeSomeSounds(Object[] objArray) {
      System.out.println ("\nMaking sounds of all array elements :
   ");
      for (int i = 0; i < objArray.length; i++) {
         if (objArray[i] instanceof Dog) {
            Dog d = (Dog) objArray[i];
            d.makeSound();
         }
         else if (objArray[i] instanceof Car) {
            Car c = (Car) objArray[i];
            c.makeSound();
         }
         else if (objArray[i] instanceof Student) {
            Student s = (Student) objArray[i];
            s.makeSound();
         }
      }
   }
```

```
  public static void main(String[] args) {

     Object[] firstArray = {new Dog(), new Car(), new Student(),
  new Car()};
     Object[] secondArray = {new Car(), new Dog(), new Student(),
  new Dog(), new Student()};

     WithoutInheritance wi = new WithoutInheritance();
     wi.makeSomeSounds(firstArray);
     wi.makeSomeSounds(secondArray);
  }
}
```

7. Although this approach also ensures that the correct `makeSound()` functionality from the respective class is accomplished, it **DOES NOT involve polymorphism**. This is because the `makeSound()` method is invoked on a different object instance (`Student`, `Car` and `Dog`) each time it is called. Compare this to the case of `WithInheritance.java`, where the method `identify()` is invoked on the same object type of the array element.

   Consider the implications of this when an additional class needs to be added. We need to add another class called `Engineer` that is a subclass of `Human` with its own `identify()` method into `WithInheritance.java`

   We can add objects from this new class into the arrays that we pass to the `ShowYourIdentity()` method; and its functionality will be achieved WITHOUT the need to modify the code in `showYourIdentity()`. Polymorphism ensures that the JVM will call the correct method if there is an `Engineer` object in the array.

   On the other hand, consider when we introduce another class called `Building` into `WithoutInheritance.java`. If we add objects from this new class into the arrays that we pass to the `makeSomeSounds()` method, we will now need to add additional code to this method as well to check for the type of this class and cast to this class in order to call the `makeSound()` method on it (in the same way as for the `Student`, `Car` and `Dog` class). If we introduce another 15 new classes into `WithoutInheritance.java`, the code for `makeSomeSounds()` will expand considerably.

   Polymorphism as a key principle in object oriented design thus significantly helps the streamlining and condensing of code that needs to be written.

8. In the case of `WithoutInheritance.java`, we cannot artificially create an inheritance relationship between the Student, Car and Dog classes in order to enjoy the advantages of polymorphic code writing. We need the **interface** mechanism in which objects of unrelated classes can still be used in a polymorphic manner.

9. An **interface specifies functionality** that needs to be accomplished, **without providing details** of how this is implemented. It declares one or more methods with no body, which are implicitly abstract and public. Once an interface is defined, any number of classes can implement it. A single class can implement any number of interfaces.

   To implement an interface, a class must provide implementations for all the methods declared within an interface. Each class is free to determine the details of its own implementation. An interface reference variable can be used to refer to any object of any class that implements that interface, and invoke any of the methods declared in the interface. The actual functionality that is executed will then depend on the class of the object being referred to.

   Interfaces are used instead of inheritance when there is certain common functionality that is shared across classes that do not belong within the same logical grouping outlined by an inheritance hierarchy.

10. There are two interfaces, `FirstInterface` and `SecondInterface` declared in `ShowInterface.java`. They also contain variables which are implicitly `public`, `final`, and `static`. These variables must be initialised when declared, otherwise a compilation error will occur. Thus, interface variables are essentially constants and cannot be further modified.

    `MyFirstClass` is defined as implementing `FirstInterface`, and therefore it must implement all the methods defined in `FirstInterface` (`getSomeThing()` and `saySomeThing()`). In addition, it can define its own methods that are unique to itself alone (`myOwnMethod()`). Notice that the statement `MIN = 20;` (attempt to change the value of the interface variable) will result in a compile time error (all interface variables are constants by default). Uncomment the statement to verify this. `MyThirdClass` implements both interfaces `FirstInterface` and `SecondInterface`, and must therefore provide definitions for all the methods declared in these two interfaces.

ShowInterface.java

```
package my.edu.utar;

interface FirstInterface {
   int getSomeThing();
   void saySomeThing(String s);
   int MIN = 50;
}

interface SecondInterface {
   void secondMethod();
   int MAX = 100;
}

class MyFirstClass implements FirstInterface {

   public void myOwnMethod() {
      System.out.println(MIN);
//    MIN = 20; // cannot do this, interface variables are final and
   static
   }
```

```java
   public int getSomeThing() {
      return 5;
   }
   public void saySomeThing(String s) {
      System.out.println("MyFirstClass : " + s);
   }
}

class MySecondClass implements FirstInterface {
   public int getSomeThing() {
      return 10;
   }
   public void saySomeThing(String s) {
      System.out.println("MySecondClass : " + s);
   }
}

class MyThirdClass implements FirstInterface, SecondInterface {
   public int getSomeThing() {
      return 15;
   }
   public void saySomeThing(String s) {
      System.out.println("MyThirdClass : " + s);
   }

   public void secondMethod() {
      System.out.println (MIN + MAX);
   }
}

public class ShowInterface {

   public static void main(String[] args) {
      FirstInterface f1 = new MyFirstClass();
      f1.saySomeThing("Hello");
//    f1.myOwnMethod(); // myOwnMethod is unique to MyFirstClass,
   not FirstInterface
      ((MyFirstClass) f1).myOwnMethod();

      f1 = new MySecondClass();
      f1.saySomeThing("Hello");
      f1 = new MyThirdClass();
//    f1.secondMethod(); // secondMethod is not defined in
   FirstInterface
      SecondInterface s1 = (SecondInterface) f1;
      s1.secondMethod();

      MyFirstClass mfc1 = new MyFirstClass();
      MyFirstClass mfc2 = new MyFirstClass();
      MySecondClass msc1 = new MySecondClass();
      MyThirdClass mtc1 = new MyThirdClass();

      FirstInterface[] interArray = {mfc1, mtc1, mfc2, msc1};
      System.out.println("Calling workWithInterfaceArray");
      workWithInterfaceArray(interArray);
   }
```

```
   public static void workWithInterfaceArray(FirstInterface[]
   interArray) {
      for (int i = 0; i < interArray.length; i++) {
         interArray[i].saySomeThing("great !");
      }
   }
}
```

11. We declare a `FirstInterface` reference variable and assign an object of class `MyFirstClass` to it. This is possible since `MyFirstClass` implements `FirstInterface`. We can invoke `saySomeThing()` on `f1` since `saySomeThing()` is defined in the `FirstInterface` variable, but we cannot invoke `myOwnMethod()` on `f1` since this method is not defined in FirstInterface (although it is defined in the object referred to by `f1`).

    If we wish to invoke `myOwnMethod()`, we have to cast the interface to the `MyFirstClass` type. The same comments apply as well to `s1.secondMethod();` (`secondMethod()` is not defined in `FirstInterface`), and we would need to cast `f1` to `SecondInterface` or `MySecondClass` first before we can invoke `secondMethod()` on it.

12. We declare an array of type `FirstInterface`, where the array elements can refer to objects of any class that implement `FirstInterface`. This array is then passed as a parameter to the `workWithInterfaceArray()` method, which then iterates through the array and calls the appropriate `saySomeThing()` method on each element of the array.

    Thus, if an array element refers to a `MyFirstClass` object, then the `saySomeThing()` method defined in the `MyFirstClass` class is executed. If the array element refers to a `MySecondClass` object, then the `saySomeThing()` method defined in the `MySecondClass` class is executed. Execute `ShowInterface` to verify this.

13. The `WithInterface.java` is a refactoring of the code in `WithoutInheritance.java` to employ interfaces so that polymorphic behaviour can be achieved. The classes `Student`, `Car` and `Dog` originally from `WithoutInheritance.java` have being renamed with an additional prefix `New`, and they now all implement a common interface `MakeSoundBehaviour` which defines the `makeSound()` method.

```
package my.edu.utar;

interface MakeSoundBehaviour {
   public void makeSound();
}
```

```java
class NewStudent implements MakeSoundBehaviour {
   public void makeSound() {
      System.out.println("I love software engineering !");
     }
}

class NewCar implements MakeSoundBehaviour  {
   public void makeSound() {
      System.out.println("Vroom vroom !");
     }
}

class NewDog implements MakeSoundBehaviour  {
   public void makeSound() {
      System.out.println("Bow wow bow wow !");
     }
}

class UseSoundFunctionality {

   MakeSoundBehaviour msb;

   public UseSoundFunctionality(MakeSoundBehaviour msb) {
      this.msb = msb;
   }

   public void workWithSounds() {
      msb.makeSound();
   }
}

public class WithInterface {

   public void makeSomeSounds(MakeSoundBehaviour[] interfaceArray) {

      System.out.println ("\nIdentifying all array elements : ");

      for (int i = 0; i < interfaceArray.length; i++) {
         interfaceArray[i].makeSound();
      }
   }

   public static void main(String[] args) {

      MakeSoundBehaviour[] firstArray = {new NewDog(), new NewCar(),
   new NewStudent(), new NewCar()};
      MakeSoundBehaviour[] secondArray = {new NewCar(), new
   NewDog(), new NewDog(), new NewStudent()};

      WithInterface wi = new WithInterface();
      wi.makeSomeSounds(firstArray);
      wi.makeSomeSounds(secondArray);

      System.out.println ("\Different functionality through
   interfaces");
```

```
        MakeSoundBehaviour msb = new NewDog();
        UseSoundFunctionality usf1 = new UseSoundFunctionality(msb);
        usf1.workWithSounds();

        MakeSoundBehaviour msb2 = new NewCar();
        UseSoundFunctionality usf2 = new UseSoundFunctionality(msb2);
        usf2.workWithSounds();

        MakeSoundBehaviour msb3 = new NewStudent();
        UseSoundFunctionality usf3 = new UseSoundFunctionality(msb3);
        usf3.workWithSounds();
    }
}
```

Two arrays of various objects that implement the `MakeSoundBehaviour` interface are passed to the `makeSomeSounds()` method; which iterates through them and calls the appropriate `makeSound()` method on these objects.

14. In the class `WithInterface`, the array of interface reference variables used in a polymorphic manner is directly provided to `makeSomeSounds()` as a method parameter. **Another way** to work with interface reference variables in a polymorphic manner is to include them as instance variables of a class; such as in `UseSoundFunctionality`. Here **`msb`** is an instance variable of the `MakeSoundBehaviour` interface type which is initialized via the constructor of the class. The `workWithSounds()` method then calls an appropriate method on this instance variable.

    We instantiate 3 different objects from `UseSoundFunctionality`; and we pass it an object from the 3 different classes that implement `MakeSoundBehaviour` through its constructor. This is subsequently used to initialize its `msb` instance variable which is then used in the `workWithSounds()` method.

Java Packages

java.lang

Provides classes that are fundamental to the design of the Java programming language. This package contains classes as listed.

Class	**Description**
Boolean	The Boolean class wraps a value of the primitive type `boolean` in an object.
Byte	The `Byte` class wraps a value of primitive type `byte` in an object.
Character	The `Character` class wraps a value of the primitive type `char` in an object.
Character.Subset	Instances of this class represent particular subsets of the Unicode character set.
Character.UnicodeBlock	A family of character subsets representing the character blocks in the Unicode specification.
Class<T>	Instances of the class `Class` represent classes and interfaces in a running Java application.
ClassLoader	A class loader is an object that is responsible for loading classes.
ClassValue<T>	Lazily associate a computed value with (potentially) every type.
Compiler	The `Compiler` class is provided to support Java-to-native-code compilers and related services.
Double	The `Double` class wraps a value of the primitive type `double` in an object.
Enum<E extends Enum<E>>	This is the common base class of all Java language enumeration types.
Float	The `Float` class wraps a value of primitive type `float` in an object.
InheritableThreadLocal<T>	This class extends `ThreadLocal` to provide inheritance of values from parent thread to child thread: when a child thread is created, the child receives initial values for all inheritable thread-local variables for which the parent has values.
Integer	The `Integer` class wraps a value of the primitive type `int` in an object.
Long	The `Long` class wraps a value of the primitive type `long` in an object.
Math	The class `Math` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Number	The abstract class `Number` is the superclass of platform classes representing numeric values that are convertible to the primitive types `byte`, `double`, `float`, `int`, `long`, and `short`.
Object	Class `Object` is the root of the class hierarchy.
Package	`Package` objects contain version information about the implementation and specification of a Java package.
Process	The `ProcessBuilder.start()` and `Runtime.exec` methods create a native process and return an instance of a subclass of `Process` that can be used to control the process and obtain information about it.
ProcessBuilder	This class is used to create operating system processes.
ProcessBuilder.Redirect	Represents a source of subprocess input or a destination of subprocess output.
Runtime	Every Java application has a single instance of class `Runtime` that allows the application to interface with the environment in which the application is running.
RuntimePermission	This class is for runtime permissions.
SecurityManager	The security manager is a class that allows applications to implement a security policy.
Short	The `Short` class wraps a value of primitive type `short` in an object.
StackTraceElement	An element in a stack trace, as returned by `Throwable.getStackTrace()`.
StrictMath	The class `StrictMath` contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
String	The `String` class represents character strings.
StringBuffer	A thread-safe, mutable sequence of characters.
StringBuilder	A mutable sequence of characters.
System	The `System` class contains several useful class fields and methods.
Thread	A *thread* is a thread of execution in a program.
ThreadGroup	A thread group represents a set of threads.
ThreadLocal<T>	This class provides thread-local variables.
Throwable	The `Throwable` class is the superclass of all errors and exceptions in the Java language.
Void	The `Void` class is an uninstantiable placeholder class to hold a reference to the `Class` object representing the Java keyword void.

java.io
Provides for system input and output through data streams, serialization and the file system.

Classes in this package are as listed.

Class	Description
BufferedInputStream	A `BufferedInputStream` adds functionality to another input stream-namely, the ability to buffer the input and to support the `mark` and `reset` methods.
BufferedOutputStream	The class implements a buffered output stream.
BufferedReader	Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
BufferedWriter	Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
ByteArrayInputStream	A `ByteArrayInputStream` contains an internal buffer that contains bytes that may be read from the stream.
ByteArrayOutputStream	This class implements an output stream in which the data is written into a byte array.
CharArrayReader	This class implements a character buffer that can be used as a character-input stream.
CharArrayWriter	This class implements a character buffer that can be used as an Writer.
Console	Methods to access the character-based console device, if any, associated with the current Java virtual machine.
DataInputStream	A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.
DataOutputStream	A data output stream lets an application write primitive Java data types to an output stream in a portable way.
File	An abstract representation of file and directory pathnames.
FileDescriptor	Instances of the file descriptor class serve as an opaque handle to the underlying machine-specific structure representing an open file, an open socket, or another source or sink of bytes.
FileInputStream	A `FileInputStream` obtains input bytes from a file in a file system.
FileOutputStream	A file output stream is an output stream for writing data to a `File` or to a `FileDescriptor`.
FilePermission	This class represents access to a file or directory.
FileReader	Convenience class for reading character files.
FileWriter	Convenience class for writing character files.

FilterInputStream	A `FilterInputStream` contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
FilterOutputStream	This class is the superclass of all classes that filter output streams.
FilterReader	Abstract class for reading filtered character streams.
FilterWriter	Abstract class for writing filtered character streams.
InputStream	This abstract class is the superclass of all classes representing an input stream of bytes.
InputStreamReader	An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified `charset`.
LineNumberInputStream	Deprecated This class incorrectly assumes that bytes adequately represent characters.
LineNumberReader	A buffered character-input stream that keeps track of line numbers.
ObjectInputStream	An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream.
ObjectInputStream.GetField	Provide access to the persistent fields read from the input stream.
ObjectOutputStream	An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.
ObjectOutputStream.PutField	Provide programmatic access to the persistent fields to be written to ObjectOutput.
ObjectStreamClass	Serialization's descriptor for classes.
ObjectStreamField	A description of a Serializable field from a Serializable class.
OutputStream	This abstract class is the superclass of all classes representing an output stream of bytes.
OutputStreamWriter	An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified `charset`.
PipedInputStream	A piped input stream should be connected to a piped output stream; the piped input stream then provides whatever data bytes are written to the piped output stream.
PipedOutputStream	A piped output stream can be connected to a piped input stream to create a communications pipe.
PipedReader	Piped character-input streams.
PipedWriter	Piped character-output streams.
PrintStream	A `PrintStream` adds functionality to another output stream, namely the ability to print representations of various data values conveniently.

PrintWriter	Prints formatted representations of objects to a text-output stream.
PushbackInputStream	A `PushbackInputStream` adds functionality to another input stream, namely the ability to "push back" or "unread" one byte.
PushbackReader	A character-stream reader that allows characters to be pushed back into the stream.
RandomAccessFile	Instances of this class support both reading and writing to a random access file.
Reader	Abstract class for reading character streams.
SequenceInputStream	A `SequenceInputStream` represents the logical concatenation of other input streams.
SerializablePermission	This class is for Serializable permissions.
StreamTokenizer	The `StreamTokenizer` class takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time.
StringBufferInputStream	Deprecated This class does not properly convert characters into bytes.
StringReader	A character stream whose source is a string.
StringWriter	A character stream that collects its output in a string buffer, which can then be used to construct a string.
Writer	Abstract class for writing to character streams.

java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Classes in this package are listed in the following table.

Class	**Description**
AbstractCollection<E>	This class provides a skeletal implementation of the `Collection` interface, to minimize the effort required to implement this interface.
AbstractList<E>	This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).
AbstractMap<K,V>	This class provides a skeletal implementation of the `Map` interface, to minimize the effort required to implement this interface.
AbstractMap.SimpleEntry<K,V>	An Entry maintaining a key and a value.
AbstractMap.SimpleImmutableEntry<K,V>	An Entry maintaining an immutable key and value.
AbstractQueue<E>	This class provides skeletal implementations of some `Queue` operations.
AbstractSequentialList<E>	This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).
AbstractSet<E>	This class provides a skeletal implementation of the `Set` interface to minimize the effort required to implement this interface.
ArrayDeque<E>	Resizable-array implementation of the `Deque` interface.
ArrayList<E>	Resizable-array implementation of the `List` interface.
Arrays	This class contains various methods for manipulating arrays (such as sorting and searching).
Base64	This class consists exclusively of static methods for obtaining encoders and decoders for the Base64 encoding scheme.
Base64.Decoder	This class implements a decoder for decoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.

Base64.Encoder	This class implements an encoder for encoding byte data using the Base64 encoding scheme as specified in RFC 4648 and RFC 2045.
BitSet	This class implements a vector of bits that grows as needed.
Calendar	The Calendar class is an abstract class that provides methods for converting between a specific instant in time and a set of calendar fields such as YEAR, MONTH, DAY_OF_MONTH, HOUR, and so on, and for manipulating the calendar fields, such as getting the date of the next week.
Calendar.Builder	Calendar.Builder is used for creating a Calendar from various date-time parameters.
Collections	This class consists exclusively of static methods that operate on or return collections.
Currency	Represents a currency.
Date	The class Date represents a specific instant in time, with millisecond precision.
Dictionary<K,V>	The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values.
DoubleSummaryStatistics	A state object for collecting statistics such as count, min, max, sum, and average.
EnumMap<K extends Enum<K>,V>	A specialized Map implementation for use with enum type keys.
EnumSet<E extends Enum<E>>	A specialized Set implementation for use with enum types.
EventListenerProxy<T extends EventListener>	An abstract wrapper class for an EventListener class which associates a set of additional parameters with the listener.
EventObject	The root class from which all event state objects shall be derived.
FormattableFlags	FomattableFlags are passed to the Formattable.formatTo() method and modify the output format for Formattables.
Formatter	An interpreter for printf-style format strings.
GregorianCalendar	GregorianCalendar is a concrete subclass of Calendar and provides the standard calendar system used by most of the world.
HashMap<K,V>	Hash table based implementation of the Map interface.

HashSet<E>	This class implements the `Set` interface, backed by a hash table (actually a `HashMap` instance).
Hashtable<K,V>	This class implements a hash table, which maps keys to values.
IdentityHashMap<K,V>	This class implements the `Map` interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values).
IntSummaryStatistics	A state object for collecting statistics such as count, min, max, sum, and average.
LinkedHashMap<K,V>	Hash table and linked list implementation of the `Map` interface, with predictable iteration order.
LinkedHashSet<E>	Hash table and linked list implementation of the `Set` interface, with predictable iteration order.
LinkedList<E>	Doubly-linked list implementation of the `List` and `Deque` interfaces.
ListResourceBundle	`ListResourceBundle` is an abstract subclass of `ResourceBundle` that manages resources for a locale in a convenient and easy to use list.
Locale	A `Locale` object represents a specific geographical, political, or cultural region.
Locale.Builder	`Builder` is used to build instances of `Locale` from values configured by the setters.
Locale.LanguageRange	This class expresses a *Language Range* defined in RFC 4647 Matching of Language Tags.
LongSummaryStatistics	A state object for collecting statistics such as count, min, max, sum, and average.
Objects	This class consists of `static` utility methods for operating on objects.
Observable	This class represents an observable object, or "data" in the model-view paradigm.
Optional<T>	A container object which may or may not contain a non-null value.
OptionalDouble	A container object which may or may not contain a `double` value.
OptionalInt	A container object which may or may not contain a `int` value.
OptionalLong	A container object which may or may not contain a `long` value.
PriorityQueue<E>	An unbounded priority queue based on a priority heap.

Properties	The `Properties` class represents a persistent set of properties.
PropertyPermission	This class is for property permissions.
PropertyResourceBundle	`PropertyResourceBundle` is a concrete subclass of `ResourceBundle` that manages resources for a locale using a set of static strings from a property file.
Random	An instance of this class is used to generate a stream of pseudorandom numbers.
ResourceBundle	Resource bundles contain locale-specific objects.
ResourceBundle.Control	`ResourceBundle.Control` defines a set of callback methods that are invoked by the `ResourceBundle.getBundle` factory methods during the bundle loading process.
Scanner	A simple text scanner which can parse primitive types and strings using regular expressions.
ServiceLoader<S>	A simple service-provider loading facility.
SimpleTimeZone	`SimpleTimeZone` is a concrete subclass of `TimeZone` that represents a time zone for use with a Gregorian calendar.
Spliterators	Static classes and methods for operating on or creating instances of `Spliterator` and its primitive specializations `Spliterator.OfInt`, `Spliterator.OfLong`, and `Spliterator.OfDouble`.
Spliterators.AbstractDoubleSpliterator	An abstract `Spliterator.OfDouble` that implements `trySplit` to permit limited parallelism.
Spliterators.AbstractIntSpliterator	An abstract `Spliterator.OfInt` that implements `trySplit` to permit limited parallelism.
Spliterators.AbstractLongSpliterator	An abstract `Spliterator.OfLong` that implements `trySplit` to permit limited parallelism.
Spliterators.AbstractSpliterator<T>	An abstract `Spliterator` that implements `trySplit` to permit limited parallelism.
SplittableRandom	A generator of uniform pseudorandom values applicable for use in (among other contexts) isolated parallel computations that may generate subtasks.

Stack<E>	The `Stack` class represents a last-in-first-out (LIFO) stack of objects.
StringJoiner	`StringJoiner` is used to construct a sequence of characters separated by a delimiter and optionally starting with a supplied prefix and ending with a supplied suffix.
StringTokenizer	The string tokenizer class allows an application to break a string into tokens.
Timer	A facility for threads to schedule tasks for future execution in a background thread.
TimerTask	A task that can be scheduled for one-time or repeated execution by a Timer.
TimeZone	`TimeZone` represents a time zone offset, and also figures out daylight savings.
TreeMap<K,V>	A Red-Black tree based `NavigableMap` implementation.
TreeSet<E>	A `NavigableSet` implementation based on a `TreeMap`.
UUID	A class that represents an immutable universally unique identifier (UUID).
Vector<E>	The `Vector` class implements a growable array of objects.
WeakHashMap<K,V>	Hash table based implementation of the `Map` interface, with *weak keys*.