Core Java 8

Lesson 11 : Abstract Classes and Interfaces

Capgemini

Lesson Objectives

After completing this lesson, participants will be able to:

- Understand concept of Abstract classes and Interfaces
- Default and static methods in interface
- Differentiate between abstract classes and interfaces
- Anonymous classes
- Implement Runtime polymorphism

Abstract Class

Provides common behavior across a set of subclasses

Not designed to have instances that work

One or more methods are declared but may not be defined, these methods are abstract methods.

Abstract method do not have implementation

Advantages:

- Code reusability
- Help at places where implementation is not available

Abstract Class (cont..)

Declare any class with even one method as abstract as *abstract*

Cannot be instantiated

Cannot use *Abstract* modifier for:

- Constructors
- Static methods

Abstract class' subclasses should implement all methods or declare themselves as *abstract*

Can have concrete methods also

Demo

Execute the Executor.java program

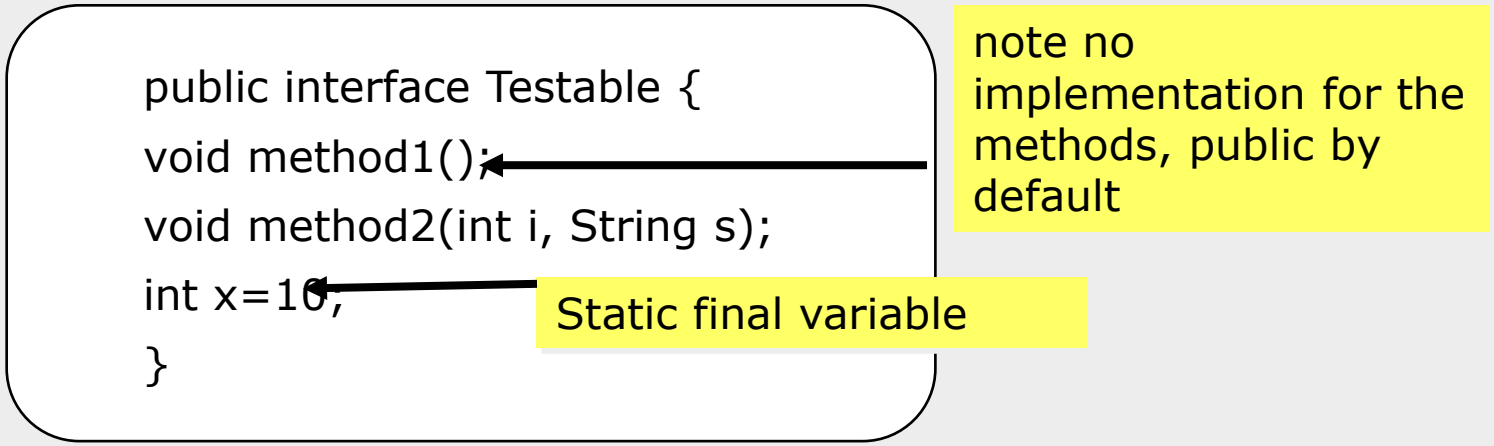
Interface

- Special kind of class which consist of only the constants and the method signatures.
- Approach also known as “programming by contract”.
- It’s essentially a collection of constants and abstract methods.
- It is used via the keyword “implements”. Thus, a class can be declared as follows:

```
class MyClass implements MyInterface{
   ...
}
```

What is Interface?

A Java interface definition looks like a class definition that has only abstract methods, although the abstract keyword need not appear in the definition



Declaring and Using Interfaces

```
public interface SimpleCalc {
        int add(int a, int b);
        int i = 10;
 }
//Interfaces are to be implemented.
class Calc implements SimpleCalc {
        int add(int a, int b){
                return a + b;
        }
}
```

abstract method

By default is public, static and final

Demo

Execute the Interface Implementation.java program

Default Methods

Starting from Java SE 8, interfaces can define default methods

A default method in an interface is a method with implementation

Use "default " keyword in method signature to make it default.

```
interface xyz {
    default return-type method-
name(argument-list) {
        ----------------
        ----------------
    }
}
```

A class which implements the interface doesn't need to implement default methods

Static Methods

Along with the default methods an Interface can also have static methods

The syntax of static method is similar to default method, where static keyword will replace default

```
interface xyz {
    static return-type method-name(argument-list)
{
        ------------------
        ------------------
    }
}
```

Demo

Interface with default and static methods

Interface - Rules

Methods other than default and static in an interface are always public and abstract.

Static methods in interface are always public .

Data members in a interface are always public, static and final.

Interfaces can extend other interfaces.

A class can inherit from a single base class, but can implement multiple interfaces.

Abstract Classes and Interfaces

Abstract classes	**Interfaces**
Abstract classes are used only when there is a “is-a” type of relationship between the classes.	Interfaces can be implemented by classes that are not related to one another.
You cannot extend more than one abstract class.	You can extend more than one interface.
Abstract class can contain abstract as well as implemented methods.	Interfaces contain only abstract, default and static methods.
With abstract classes, you grab away each class’s individuality.	With Interfaces, you merely extend each class’s functionality.

Inner Classes

Java inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax :

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Method Local Inner Classes

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted to the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

Method Local Inner Classes

```
public class OuterClass {
   public void display(){
      int num = 23;
      class Inner{
         public void print() {
            System.out.println("This is method inner class "+num);
         }
      }
      Inner obj = new Inner();
      obj.print();
   }
   public static void main(String args[]){
      OuterClass outer = new OuterClass();
      outer.display();
   }
}
```

Anonymous Classes

A class that has no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

- Class (may be abstract or concrete).
- Interface

It is an inner **class** without a name and for which only a single object is created.
An **anonymous** inner **class** can be useful when making an instance of an object with certain "extras" such as overloading methods of a **class** or interface, without having to actually subclass a **class**.

Anonymous Classes

```
AnonymousInner an_inner = new AnonymousInner() {
   public void my_method() {
      ........
      ........
   }
};
```

Anonymous Classes

```
abstract class AnonymousInner {
   public abstract void mymethod();
}
public class Outer_class {


   public static void main(String args[]) {
      AnonymousInner inner = new AnonymousInner() {
         public void mymethod() {
            System.out.println("This is an example of anonymous inner class");
         }};
      inner.mymethod();
   }}
```

Runtime Polymorphism

Runtime polymorphism enables a method can do different things based on the object used for invoking method at runtime

Runtime polymorphism is implemented by doing method overriding

```
class Parent {
        public String sayHello() {
            return "Hello from
Parent";
        }
}
class Child extends Parent {
        public String sayHello() {
            return "Hello from
Child";
        }
}
```

```
Parent object = new
Child();
object.sayHello();
```

Accessing Implementations through Interface Reference

```
class sample implements TestInterface {
// Implement Callback's interface
public void interfacemthod() {
        System.out.println("From interface method"); }
public void noninterfacemthod() {
        System.out.println("From interface method"); }
 }
```

```
class Test {
   public static void main(String args[]) {
        TestInterface t = new sample();
         t.interfacemethod()      //valid
         t.noninterfacemethod()       //invalid }
```

Demo

Runtime polymorphism

Lab

Lab 5: Abstract classes and Interfaces

Summary

In this lesson, you have learnt about:

- Abstract class
- Interfaces
- default methods
- static methods on Interface
- Runtime Polymorphism

Review Question

Question 1: All variables in an interface are :

- **Option 1:** Constant instance variables
- **Option 2:** Static and final
- **Option 3:** Constant instance variables

Question 2: Will this code throw a compilation error?

```
interface sample
{
  int x;
}
```

- **Option 1:** True
- **Option 2:** False