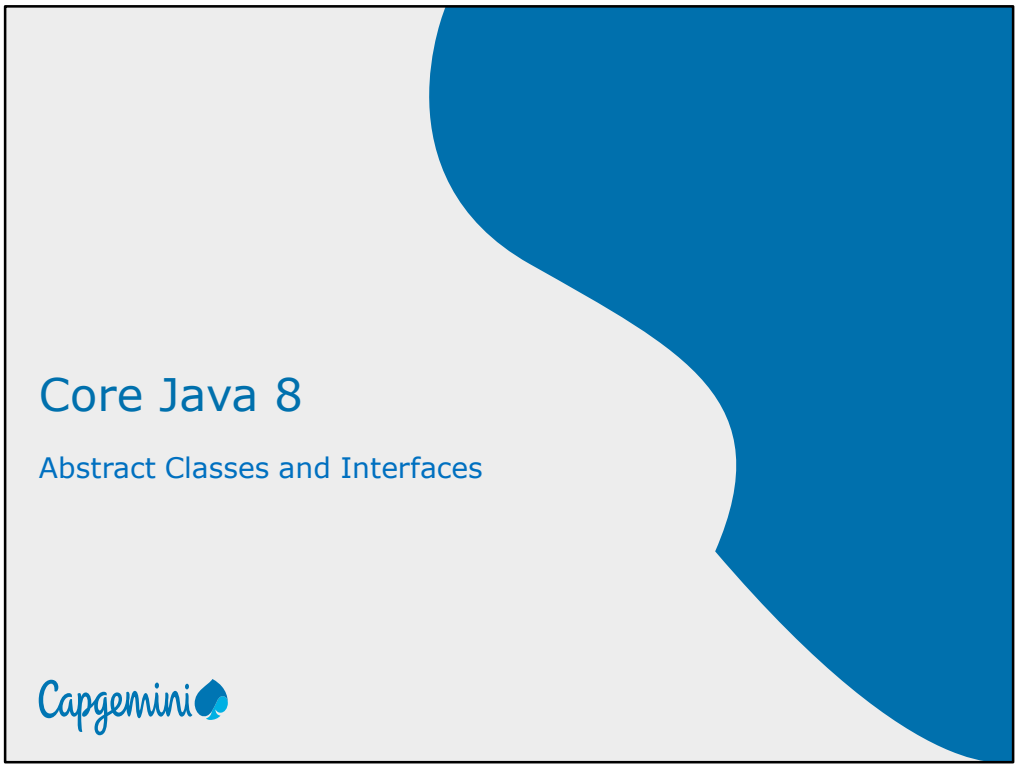

Core Java 8
Abstract Classes and Interfaces
Capgemini

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

Lesson Outline:

11.1: Abstract class
11.2: Interfaces
11.3: default methods
11.4: static methods on Interface
11.5: Abstract class Vs Interface
11.6: Anonymous Classes
11.7: Runtime Polymorphism
11.8: Best Practices

Abstract Classes
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

Example of Abstract class – Shape Hierarchy –
You have two hierarchies : Point-Circle-Cylinder and Line-Square-Cube
Here common method is area() – if I create an array that contains the objects of all these classes. How can I do it generically ?
Can Point p = new Line() be valid ? of course not, no inheritance !!!!
So, I force a super class Shape which would contain the method area(). Now what implementation I am going to write in that method. I do not know, at runtime whose area() is going to be called. So, the information that I do not know I put it as “abstract”.

Important points about abstract class :
- You cannot create object of abstract class : why ?
  For example, if a surgeon know how to perform an operation but he does not know how to stitch back the cut stomach will I allow him to touch me?
- An abstract class can contain concrete methods.
- An abstract class may not contain any abstract methods.
- In Java a pure virtual method is called as abstract method.

```
abstract class Shape{
    public abstract float area();
}
```

Abstract Classes

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

Example of Abstract modifier:

```
abstract class Shape{
    abstract void draw();        // observe : no implementation
}

class Rect extends Shape{
     void draw(){      // draw() implemented in subclass Rect
           System.out.println("Drawing a rectangle");
      }
     public static void main(String args[]){
          Shape r1 = new Rect();
          r1.draw();
    }
}
```

Output :
Drawing a rectangle

Abstract Classes
Demo

Execute the Executor.java program

The example shows how to declare and use an interface

Interfaces
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{
   ...
}
```

Interface:

You may come across a situation, in which you want to have different implementations of a method in different classes and delay the decision on which implementation of a method to execute until runtime. In java, the class where the method is defined must be present at compile time so that the compiler can check the signature of the method to ensure that the method call is legitimate. All the classes that could possibly be called for the aforementioned method need to share a common super class, so that the method can be defined in the super class and overridden by the individual subclasses. If you want to force every subclass to have its own implementation of the method, the method can be defined as an abstract one. Chances are you will want to move the method definition higher and higher up the inheritance hierarchy, so that more and more classes can override the same method.

Because of single inheritance, any Java class has only a single super class. It inherits variables and methods from all superclasses above it in the hierarchy. This makes sub-classing easier to implement and design, but it also can be restricting when you have similar behavior that must be duplicated across different branches of class hierarchy. Java solves the problem of shared behavior by using interfaces.

An interface is a collection of method signatures (without implementations) and constant values. Interfaces are used to define a protocol behavior that can be implemented by any class hierarchy. Interfaces are abstract classes that are left completely unimplemented**.** That means, no methods in the class has been implemented. Using an interface, you can specify what a class must do, but not how it does.

Interfaces

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

```
public interface Testable {
void method1();
void method2(int i, String s);
int x=10;
}
```

note no implementation for the methods, public by default

Static final variable

Interface (contd.):

Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. Additionally, interface data members are limited to **static final variables**, which means that they are constant. An object variable can be declared as an interface type, and all the constants and methods declared in the interface can be accessed from this variable. All objects, whose class types implement the interface, can then be assigned to this variable. Therefore, to solve the problem of how to decide implementation which method is to be executed at runtime, you can define an interface with the method be shared among classes. Reference to this commonly implemented method from the interface object variable will then be resolved at runtime.

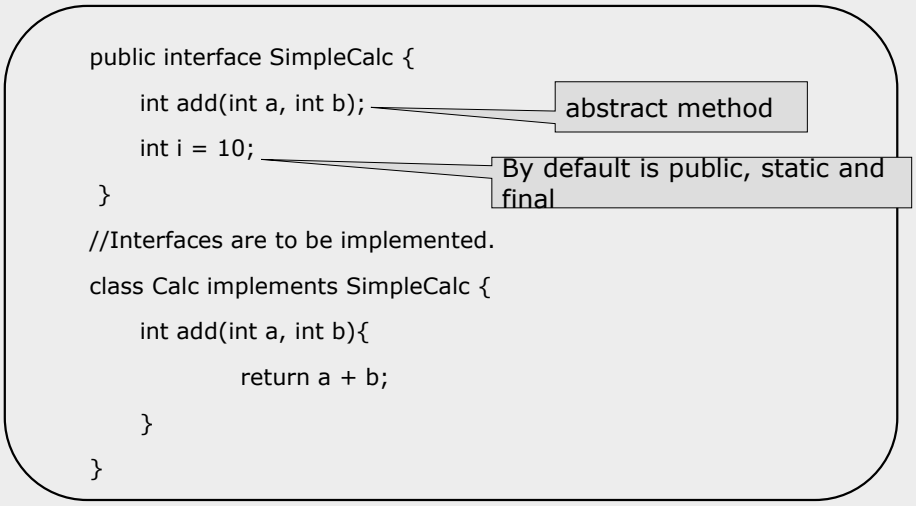
An interface definition consists of both interface declaration and interface body.

```
// Interface Declaration:
<access> interface <name>  {
        return type <method_name> ( <parameter_list> ) ;
        <type> <variable name> = <value>;
}
```

The interface declaration informs about various attributes of the interface such as its name and whether it extends to another interface.

Interfaces

Declaring and Using Interfaces



An interface declaration can have two other components:
- The public access specifier
- The list of super interfaces

While a class can only extend one other class, an interface can extend any number of interfaces, and an interface cannot extend classes. An interface inherits all constants and methods from its super interface.
The interface body contains method declarations for the methods defined within the interface and constant declarations. All constant values defined in an interface are implicitly public, static and final. Similarly, all methods declared in an interface are implicitly public and abstract. One class can implement more than one interface at a time by separating them using commas.

Note: Once, a class implements an interface it has to override all the methods in that interface; otherwise, a class has to be declared as an abstract class.

Interfaces
Demo

Execute the Interface Implementation.java program

The example shows how to declare and use an interface

Default method in Interface

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

To enhance Java API with lambda expressions, many existing interfaces needs to be modified. Adding new methods to interface leads to break the existing implementation.

To avoid this, Java has added default methods to interfaces. When you add default method to existing interface, it doesn't break the classes which implements the interface.

Note: Default interfaces concept is strictly meant for backward compatibility. It doesn't mean you create java application only with interfaces with default methods and not classes.

Rules for default methods in Interface inheritance:

While extending an Interface having default methods, there are few points to ponder:

1. Child Interface can use the default method of parent interface.
2. Child Interface can re-declare the default method without default keyword to make it abstract.
3. Child Interface can override the default method by keeping the same signature as of parent interface.

static method in Interface

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)
{
        ------------------
        ------------------
   }
}
```

Both default and static interface method now allows developer to extend the functionality of existing system without breaking the code.

If you are designing the system from scratch, it is recommended not to use these features. However, these features can be handy to modify existing application to add new functionality with ease.

The default and static methods are extensively used by Java SE 8 language developers to add new methods to existing API. For example, the **forEach()** default method added to Collection API for iterating elements in collection.

Default and static
Demo

Interface with default and static methods

The example shows how to declare and use an 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.

Add the notes here.

Abstract Class vs Interface

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.

Anonymous Classes

Anonymous Classes

A class that have 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

Anonymous Classes

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

Anonymous Classes

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

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();
```

In Java, it is legal to up cast the reference of child object to parent class reference. Also in case of interfaces, you can assign reference of implementer class to an interface object.

Parent Class reference = Child Class Object;
Interface reference = Interface Implementer class object;

In above cases, parent class reference can be used to call methods in child class which are overridden. It cannot be used to access methods owned by child class.

The code snippet shown in the slide has two classes, where child class overrides the sayHello() method of parent. Therefore parent class reference is able to access the overridden method of child class.

Note: Polymorphism does not work with static methods because they are early-bound.

Runtime Polymorphism

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 }
```

Accessing Implementations through Interface Reference:

Object references can be declared which uses an interface rather than a classtype. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the "callee."
The variable **t** is declared to be of the interface type **TestInterface**, yet it was assigned an instance of sample. Although, **t** can be used to access the **interfacemethod()** method, it cannot access any other members of the sample class. An interface reference variable only has knowledge of the methods declared by its interface declaration. Thus, **t** could not be used to access **noninterfacemethod( )** since it is defined by sample but not the **TestInterface**.

Runtime Polymorphism
Demo

Runtime polymorphism

The example shows how to declare and use an interface with default and static methods.

Abstract Classes and Interfaces
Lab

Lab 5: Abstract classes and Interfaces

Add the notes here.

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