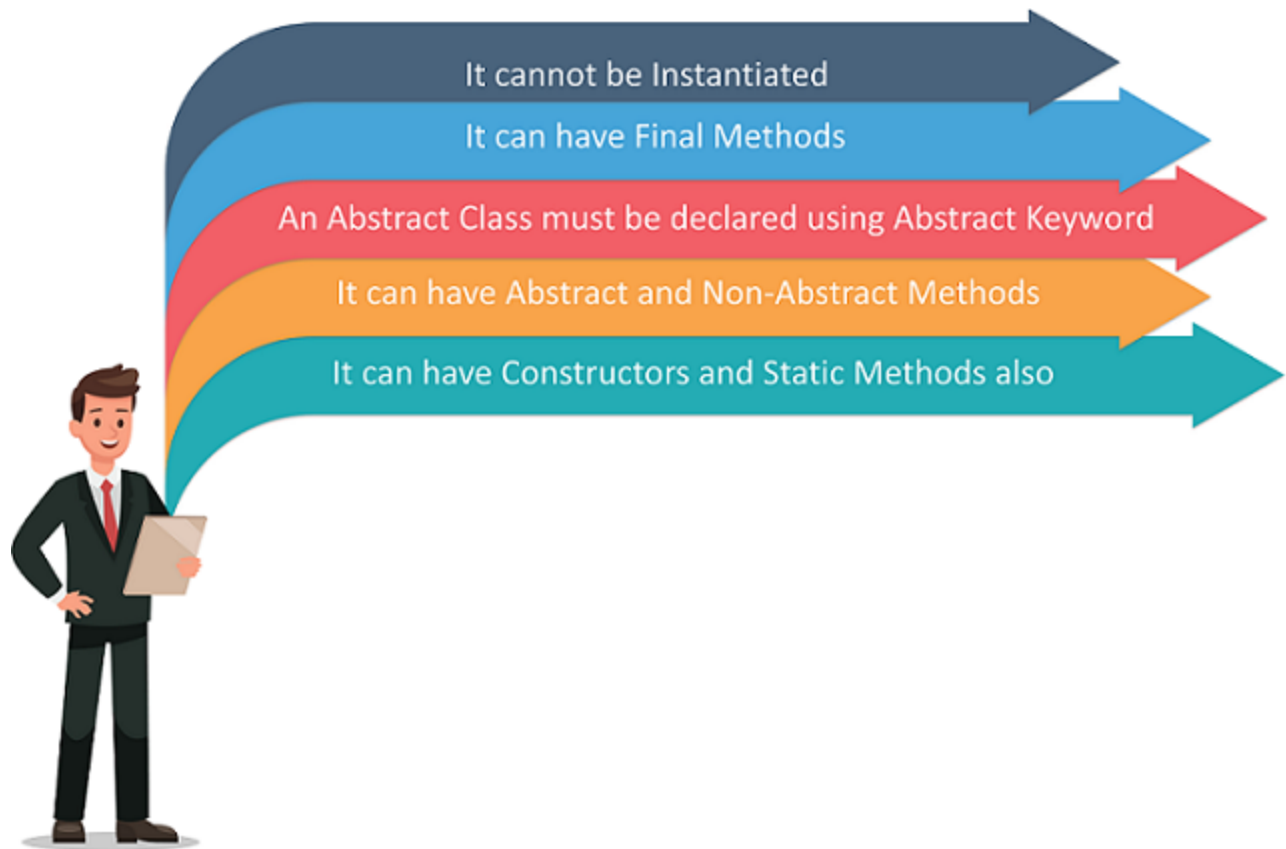


Week 06 - Abstract Classes, Interfaces and Polymorphism

Louis Botha, louis.botha@tuni.fi



A Complete Introduction to Abstract Classes in Java | Edureka

Abstraction

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either **abstract classes** or **interfaces** (we will look at interfaces later).

Here is an abstract class definition

```
abstract class Animal {
```

```
// Abstract method has no body
abstract void makeNoise();
}
```

An abstract class **can not** be instantiated, objects can't be created from an abstract class.


```
error: Animal is abstract; cannot be instantiated
    Animal animal = new Animal();
```

The `abstract` keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from in another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

```
public abstract void eat();
```

No method body!
End it with a semicolon.



Head First Java

A class that contains at least one abstract method is called an abstract class and must be declared with the `abstract` modifier on the class declaration.

An abstract class can have both abstract and regular methods:

```
abstract class Animal {
    // Abstract method has no body
    abstract void makeNoise();
    public void run() {
        System.out.println("The ANIMAL run");
    }
}
```

```

class Dog extends Animal {
    // Subclass must implement the body
    @Override
    public void makeNoise() {
        System.out.println("The DOG goes woof, woof");
    }
}

class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeNoise();
        myDog.run();
    }
}

```

```

// Output
> java Main
The DOG goes woof, woof
The ANIMAL run

```

Interface

Defining a basic interface.

```

public interface Animal {
    void makeNoise();
}

```

The interface above named `Animal`, includes a single method named `makeNoise`. Any class that **implements** the `Animal` interface must provide an implementation for a method named `makeNoise` that accepts no parameters and doesn't return a value.

You say "interface" instead of "class" here.

Interface methods are implicitly public and abstract, so typing in "public" and "abstract" is optional (in fact, it's not considered "good style" to type the words in, but we did here just to reinforce it).

All interface methods are abstract, so they **MUST** end in semicolons. Remember, they have no body!

```
public interface Pet {  
    public abstract void beFriendly();  
    public abstract void play();  
}
```

Head First Java

Below the `Dog` class **implements** the `Animal` interface and implements the `makeNoise()` method.

```
public interface Animal {  
    void makeNoise();  
}  
  
public class Dog implements Animal {  
    @Override  
    public void makeNoise() {  
        System.out.println("Woof, Woof");  
    }  
}  
  
class TestAnimal {  
    public static void main(String[] args) {  
        Dog myDog = new Dog();  
        myDog.makeNoise();  
    }  
}
```

An interface class can not be instantiated, objects can't be created from an interface class.

```
error: Animal is abstract; cannot be instantiated
    Animal animal = new Animal();
```

A class can also implement more than one interface.

```
interface Animal {
    void makeNoise();
}

interface Pet {
    void ownerCalls();
}

class Dog implements Animal, Pet {
    @Override
    public void makeNoise() {
        System.out.println("Woof, Woof");
    }

    @Override
    public void ownerCalls() {
        System.out.println("Come, Come");
    }
}

class TestAnimal {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.makeNoise();
    }
}
```

```
        myDog.ownerCalls();
    }
}
```

:attention: A class can extend only one other class, but it can implement as many interfaces as you need.

In GUI programming interfaces are very often used for handling events.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

//Creating a class that inherits from JFrame class
class MyFrame extends JFrame implements ActionListener {
    private JButton ok;

    public MyFrame() {
        // Creating a Java button
        ok = new JButton("OK");
        // Telling the button object that we have a method in
        this class
        // that can react to action events
        ok.addActionListener(this);
        // Adding the button to the frame
        add(ok);
        // Set the size of the frame
        setSize(200,100);
        // Make it visible
        setVisible(true);
    }
}
```

```
@Override
//ActionListener method that is implemented by the MyFrame
class
public void actionPerformed(ActionEvent e) {
    System.out.println("Action Performed");
}

public static void main(String [] args) {
    new MyFrame();
}
}
```

Abstract Class vs Interface

Abstract Class

1. **abstract** keyword
2. Subclasses **extends** abstract class
3. Abstract class can have implemented methods and 0 or more abstract methods
4. We can extend only one abstract class



Interface

1. **interface** keyword
2. Subclasses **implements** interfaces
3. Java 8 onwards, Interfaces can have default and static methods
4. We can implement multiple interfaces



Polymorphism

Polymorphism is the ability of an object to take on different forms during runtime, the ability of Java to use base class variables to refer to subclass objects.

edureka!



Java Polymorphism example-Edureka

In the above figure, you can see, *Man* is only one, but he takes multiple roles like – he is a dad to his child, he is an employee, a salesperson and many more. This is known as **Polymorphism**.

Inheritance lets us inherit attributes and methods from another class.

Polymorphism uses those methods to perform different tasks. To put it simply, polymorphism in Java allows us to perform the same action in many different ways.

Any Java object that can pass more than one IS-A test is polymorphic in Java.

Therefore, all the Java objects are polymorphic as it has passed the IS-A test for their own type and for the class Object.

```
class Animal {
    public void run() {
        System.out.println("The ANIMAL run");
    }
}
class Dog extends Animal {
    public void run() {
        System.out.println("The DOG run");
    }
}
class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        // Dog is an animal because of inheritance so we can pass it the argument
        doSomething(myDog);
    }

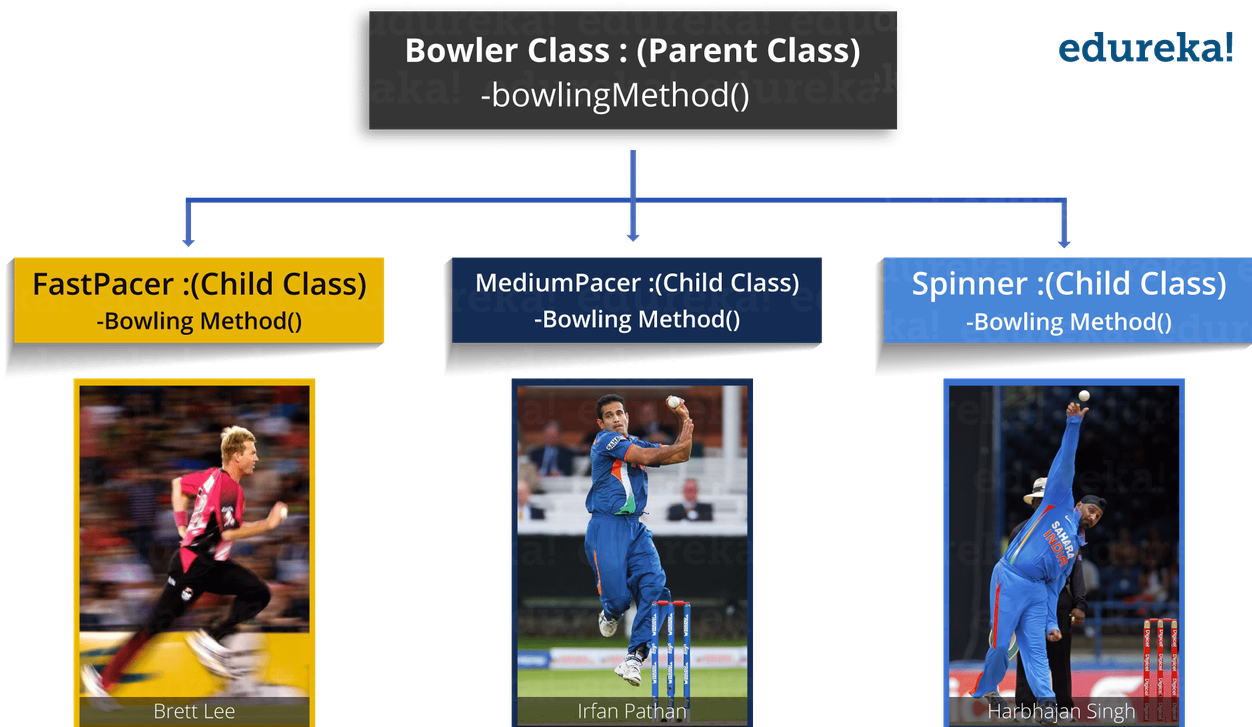
    //Here we use the base class to allow the subclass to be passed as argument
    public static void doSomething(Animal animal) {
        // Calling the Dog run method,
        // Because the parameter was a Dog object
        animal.run();
    }
}
```

```

        // We can cast the animal object back to a Dog object
        Dog dog = (Dog) animal;
        dog.run();
    }
}

```

Let's consider this real world scenario in cricket, we know that there are different types of bowlers in the sport of cricket i.e. fast bowlers, medium pace bowlers and spinners.



Polymorphism - object oriented programming - Edureka

As you can see in the above figure, there is a parent class `BowlerClass` and it has three child classes:

- `FastPacer`,
- `MediumPacer` and
- `Spinner`.

Bowler class has `bowlingMethod()` where all the child classes are inheriting this method. As we all know that a fast bowler will going to bowl differently as compared to medium pacer and spinner in terms of bowling speed, long run up and way of bowling, etc. Similarly a medium pacer's implementation of `bowlingMethod()` is also going to be different as compared to other bowlers. And same happens with spinner class.

The point of above discussion is simply that a same name tends to multiple forms. All the three classes above inherited the `bowlingMethod()` but their implementation is totally different from one another.

Polymorphism in Java is of two types:

1. Run time polymorphism
2. Compile time polymorphism

Run time polymorphism: In Java, runtime polymorphism refers to a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this, a reference variable is used to call an overridden method of a superclass at run time.

Method overriding is an example of run time polymorphism.

```
class Bowler {
    void bowlingMethod() {
        System.out.println(" bowler ");
    }
}

class FastPacer extends Bowler {
    void bowlingMethod() {
        System.out.println(" fast bowler ");
    }
}

class Spinner extends Bowler {
```

```

        void bowlingMethod() {
            System.out.println(" spin bowler ");
        }
    }

    class Main {
        public static void main(String[] args) {
            Bowler fastPacer = new FastPacer();
            fastPacer.bowlingMethod();

            Bowler spinner = new Spinner();
            spinner.bowlingMethod();
        }
    }
}

```

Compile time polymorphism: In Java, compile time polymorphism refers to a process in which a call to an overloaded method is resolved at compile time rather than at run time. Method overloading is an example of compile time polymorphism. Method Overloading is a feature that allows a class to have two or more methods having the same name but the arguments passed to the methods are different. Unlike method overriding, arguments can differ in:

1. Number of parameters passed to a method
2. Datatype of parameters
3. Sequence of datatypes when passed to a method.

```

class Adder {
    static int add(int a, int b) {
        return a+b;
    }
    static double add( double a, double b) {
        return a+b;
    }
}

```

```
public static void main(String args[]) {  
    System.out.println(Adder.add(11, 11));  
    System.out.println(Adder.add(12.3, 12.6));  
}  
}
```