Abstract Method and Abstract Class

An abstract method does not contain any body. It contains only the method header, we can say it is an incomplete method. An abstract class is a class that generally contains some abstract methods. Both the abstract class and abstract methods should be declared by using the word 'abstract'. Since abstract classes contains incomplete methods, it is not possible to estimate the total memory required to create the object. So, JVM cannot create objects to an abstract class. We should create sub classes and all the abstract methods should be implemented (body should be written) in the subclasses. Then, it is possible to create the object to the sub classes since they are complete classes.

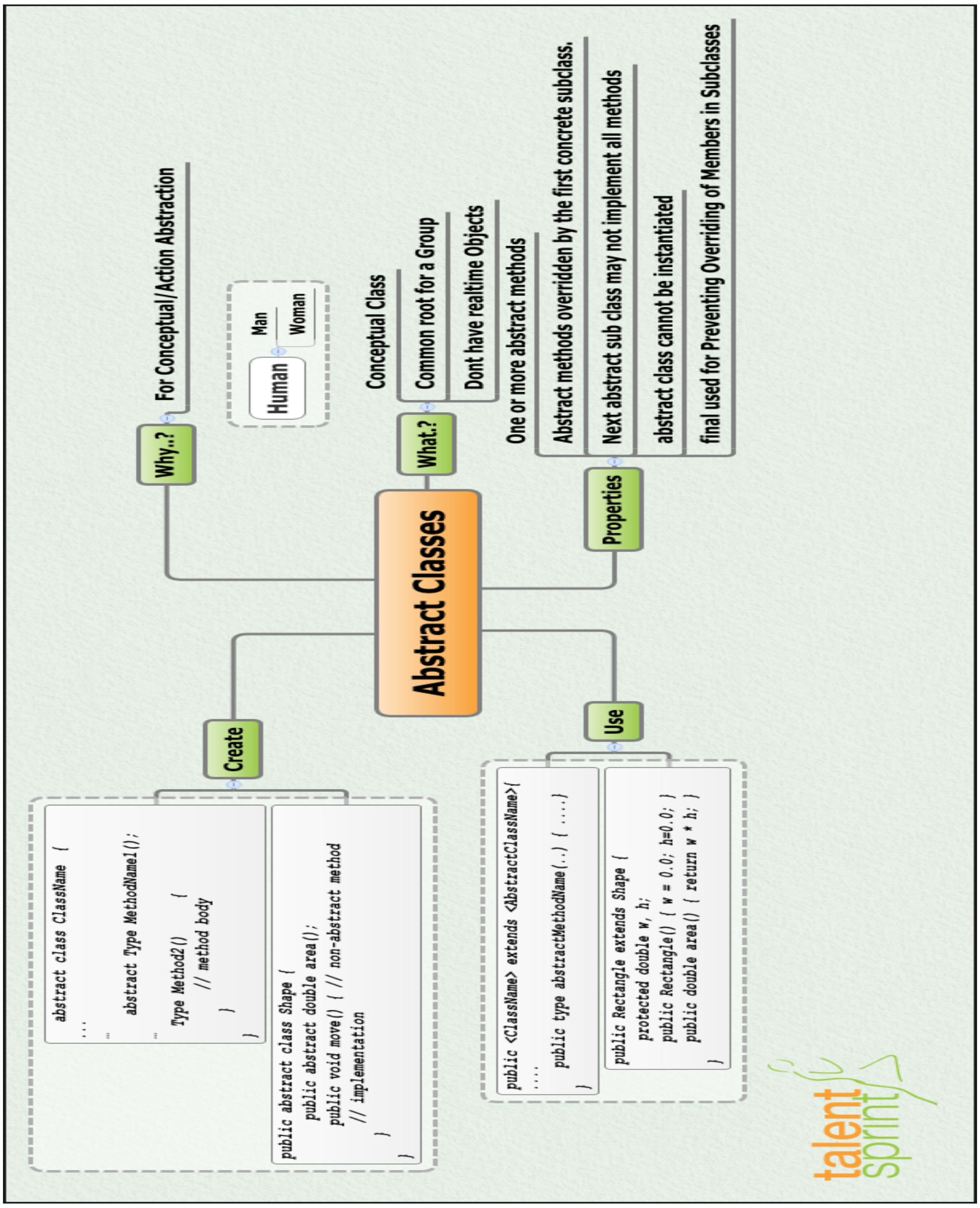
```
abstract class ClassName {
    abstract methodName();
        concreteMethod(){}
}
```

Exaxmple, Let us write abstract class with an instance variable `rate`, an abstract method `getRate()` and a concrete method `calculateBill()`

```
abstract class plan {
    protected double rate;
    public abstract void getRate();
    public void calculateBill (int units) {
        System.out.print ('' BIll amount for ''+ units + ''units :'');
        System.out.println (rate*units);
    }
}
class CommercialPlan extends Plan {
    public void getRate() {
        rate = 5.00;
    }
}
class DomesticPlan extends Plan {
    public void getRate() {
        rate = 2.60;
    }
}
class Calculate {
    public static void main(String args []) {
        Plan p;
        System.out. println (‘‘ Commercial connection : ’’);
        p = new CommercialPlan();
        p.getRate();
        p. calculateBill (250);
        System.out. println (‘‘ Domestic connection: ’’);
        p = new DomesticPlan();
        p.getRate();
        p. calculateBill (150);
    }
}
```

Important Points

- An abstract class is a class that contains 0 or more abstract methods.
- An abstract class can contain instance variables and concrete methods in addition to abstact methods.
- Abstract class and the abstract methods should be declared by using ‘`abstract`’ keyword.
- All the abstract methods of abstract class should be implemented in the sub class.
- If any abstract method is not implemented, then that sub class should be declared as ‘`abstract`’. In this case, we cannot create object to the sub class. We should create another sub class and implement the remaining abstract method there.
- We cannot create object to abstract class. But, we can create a reference of abstract class type.
- The reference of abstract class can be used to refer to objects of its sub classes.
- It is possible to derieve an abstract class as a sub class from a concrete super class.
- We cannot declare a class both **abstract** and **final** .



Abstract Classes
Why..?
For Conceptual/Action Abstraction
Human
Man
Woman
What.?
Conceptual Class
Common root for a Group
Dont have realtime Objects
Properties
One or more abstract methods
Abstract methods overridden by the first concrete subclass.
Next abstract sub class may not implement all methods
abstract class cannot be instantiated
final used for Preventing Overriding of Members in Subclasses
Create
Use
talent sprint

```
abstract class ClassName  {
...
::
        abstract Type MethodName1();
::
        Type Method2()      {
                // method body
        }
}
```

```
public abstract class Shape {
    public abstract double area();
    public void move() { // non-abstract method
    // implementation
    }
}
```

```
public <ClassName> extends <AbstractClassName>{
....
        public type abstractMethodName(..) { ....}
}
```

```
public Rectangle extends Shape {
    protected double w, h;
    public Rectangle() { w = 0.0; h=0.0; }
    public double area() { return w * h; }
}
```

Interface

An interface contains only abstract methods which are all incomplete methods. It is not possible to create an object to an interface. In this case, we can create seperate classes where we can implement all the methods of the interface. These classes are called implementation classes. Since, implentation classes will have all the methods with body, it is possible to create objects to the implementation classes. The flexibility lies in the fact that every implementation class can have its own implementation of the abstract methods of the interface.

```
    interface InterfaceName {
        variables ;
        abstract methodName();
    }
```

Example, Write a program which contains a Printer interface and its implementation classes to send text to any printer.

```
impoert java. util .*;
interface Printer {
    void print(String text);
    void disconnect();
}
class IBMPrinter implements Printer {
    public void print(String text) {
        System.out. println(text);
    }
    public void disconnect() {
        System.out. println (‘‘ printing completed’’);
        System.out. println (‘‘Disconnected from IBM Printer’’);
    }
}
class EpsonPrinter implements Printer {
    public void print(String text) {
        System.out. println(text);
    }
    public void disconnect() {
        System.out. println (‘‘ printing completed’’);
        System.out. println (‘‘Disconnected from Epson Printer’’);
    }
}
class UsePrinter {
    public static void main(String args []) {
        Scanner sc = new Scanner(System.in);
        System.out. println (“ Enter the printer name”);
        String printername = sc.next();
        try {
            Class c = Class.forName(printername);
            Printer ref = (Printer)c.newInstance();
            ref . print (“ Hello, this is printed on the printer ’’);
            ref . disconnect ();
        }
        catch(Exception ref) {
            System.out. println (“ Exception ’’ +ref);
        }
    }
}
```

Important Points

- An interface is a specification of method prototypes. This means, only method names are written in the interface without method bodies.

- An interface will have 0 or more abstract methods which are all public and abstract by default.

- An interface can have variables which are public, static and final by default. This means all the variables of the interface are constants.

- None of the methods in interface can be private, protected or static.

- We cannot create an object to an interface, but we can create a reference of interface type.

- All the methods of interface should be implemented in its implementation classes. If any method is not implemented, then that implementation class should be declared as ‘abstract’

- Interface reference can refer to the objects of its implementation classes.

- When an interface is written, any third party vendor can provide implementation classes to it.

- An interface can extend another interface.

- An interface cannot implement another interface.
- It is possible to write a class within an interface.
- Interface forces the implementation classes to implement all of its methods compulsory. Java compiler checks whether all the methods are implemented in the implementation classes or not.
- A class can implement (not extend) multiple interfaces.

```
class MyClass implements Interface1, Interface2
class MyClass extends Class1 implements Interface1, Interface2
```

Multiple Inheritance Using Interfaces

In multiple inheritance, sub classes are derived from multiple super classes. If two super classes have same name for their class members (variables and methods) then which member is inherited into the sub class is the main confusion in multiple inheritance. This is the reason, Java does not support the concept of multiple inheritance. This confusion is reduced by using mutiple interfaces to achieve multiple inheritance. For exmaple

```
interface Interface1 {
    int x = 20; // public static final
    void method(); // public abstract
}
interface Interface2 {
    int x = 30;
    void method();
}
```

And there is an implementation class MyClass as :

```
class MyClass implements Interface1, Interface2
```

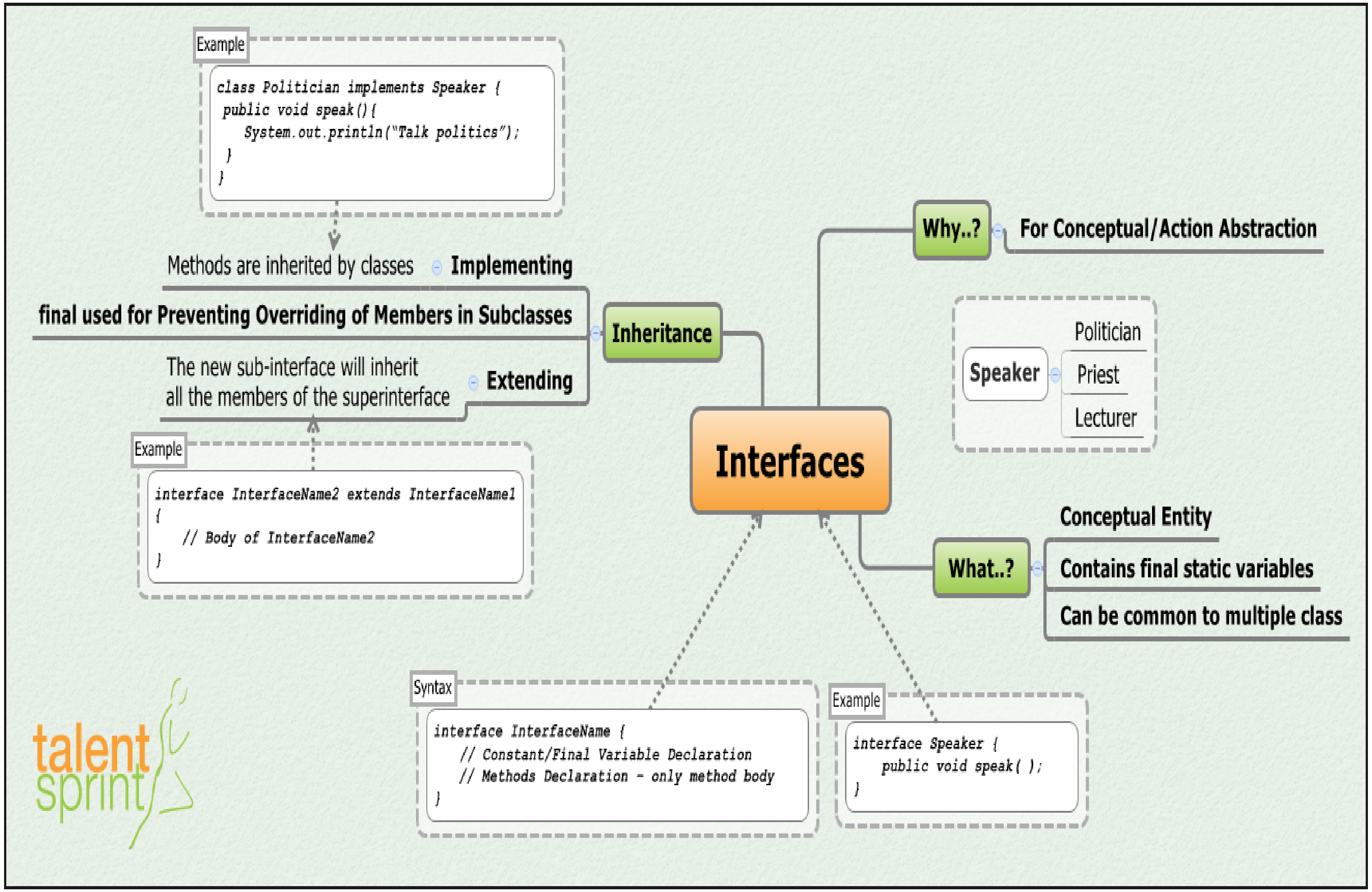
Now there is no confusion to refer to any of the members of the interfaces from MyClass. For example, to refer to interface Interface1 and Interface2 we can write :

```
Interface1 .x
Interface2 .x
```

Similarly there will not be any confusion regarding which method is available to the implementation class.

Example Write a program to illustrate how to achieve multiple inheritance using multiple interfaces.

```
interface Father {
    float ht = 6.2f;
    void height();
}
interface Mother {
    float ht = 5.8f;
    void height();
}
class child implements Father, Mother {
    public void height() {
        float ht = (Father.ht + Mother.ht) / 2;
        System.out.println ("Child's height " +ht);
    }
}
class MultipleInheritance {
    public static void main(String args[]) {
        child ch = new child();
        ch.height();
    }
}
```



Interfaces

Why..?

- For Conceptual/Action Abstraction

Speaker

- Politician
- Priest
- Lecturer

What..?

- Conceptual Entity
- Contains final static variables
- Can be common to multiple class

Syntax

```
interface InterfaceName {
    // Constant/Final Variable Declaration
    // Methods Declaration - only method body
}
```

Example

```
interface Speaker {
    public void speak( );
}
```

Inheritance

- Implementing
  - Methods are inherited by classes
- final used for Preventing Overriding of Members in Subclasses
- Extending
  - The new sub-interface will inherit all the members of the superinterface

Example

```
class Politician implements Speaker {
 public void speak(){
    System.out.println("Talk politics");
 }
}
```

Example

```
interface InterfaceName2 extends InterfaceName1
{
    // Body of InterfaceName2
}
```

Wrapper Classes

Wrapper classes are used to convert any data type into an object. The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language. Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class. The wrapper classes are part of the java.lang package, which is imported by default into all Java programs.

Integer is a wrapper class of the primitive *int*

Character is a wrapper class of the primitive *char*

Double is a wrapper class of the primitive *double*

Converting Primitive Types to Objects (Wrapper) and the vise versa

Primitive type	Primitive to Object	Object to Primitive
double d = 5.0;	Double aDouble = new Double(d);	double r = aDouble.doubleValue();
int i = 5	Integer dataCount = new Integer(i);	int newCount = dataCount.intValue();

A common translation you need in programs is converting a String to a numeric type, such as an **int** (Object –>primitive).

```
String pennsylvania = "65000";
int penn = Integer.parseInt(pennsylvania);
```

Primitive Types and their Wrapper Classes

The following table lists the primitive types and the corresponding wrapper classes:

Primitive type	Wrapper Class
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

Converting Primitive Types to Strings

valueOf() method of **String** class is used to convert numerical values to strings.

```
int i = 1;
double d = 5.0;
String dStr = String.valueOf(d);
String iStr = String.valueOf(i);
```