Abstract class and abstract methods

Abstract class means class cannot create its object and abstract method means method cannot have definition.

```
abstract class A
{abstract void show ();
}
A  a1 = new A(); //it is not allowed
```

Q. Why use the abstract method and abstract classes ?

1) abstract class is used for achieve abstraction

2) abstract method is used for achieve dynamic polymorphism

Q. What is the abstraction ?

abstraction means to hide the implementation detail from end user at designing level called as abstraction. Means in abstraction just we provide the prototype of the work we not provide the detail description about the task or work. We write its implementation part or logical part or discretional part where we want to implement it called as abstraction

There are two ways to achieve abstraction in java

i) Using abstract class

ii) Using interface

If we not write the logic of abstract method so where we can write the logic of abstract?

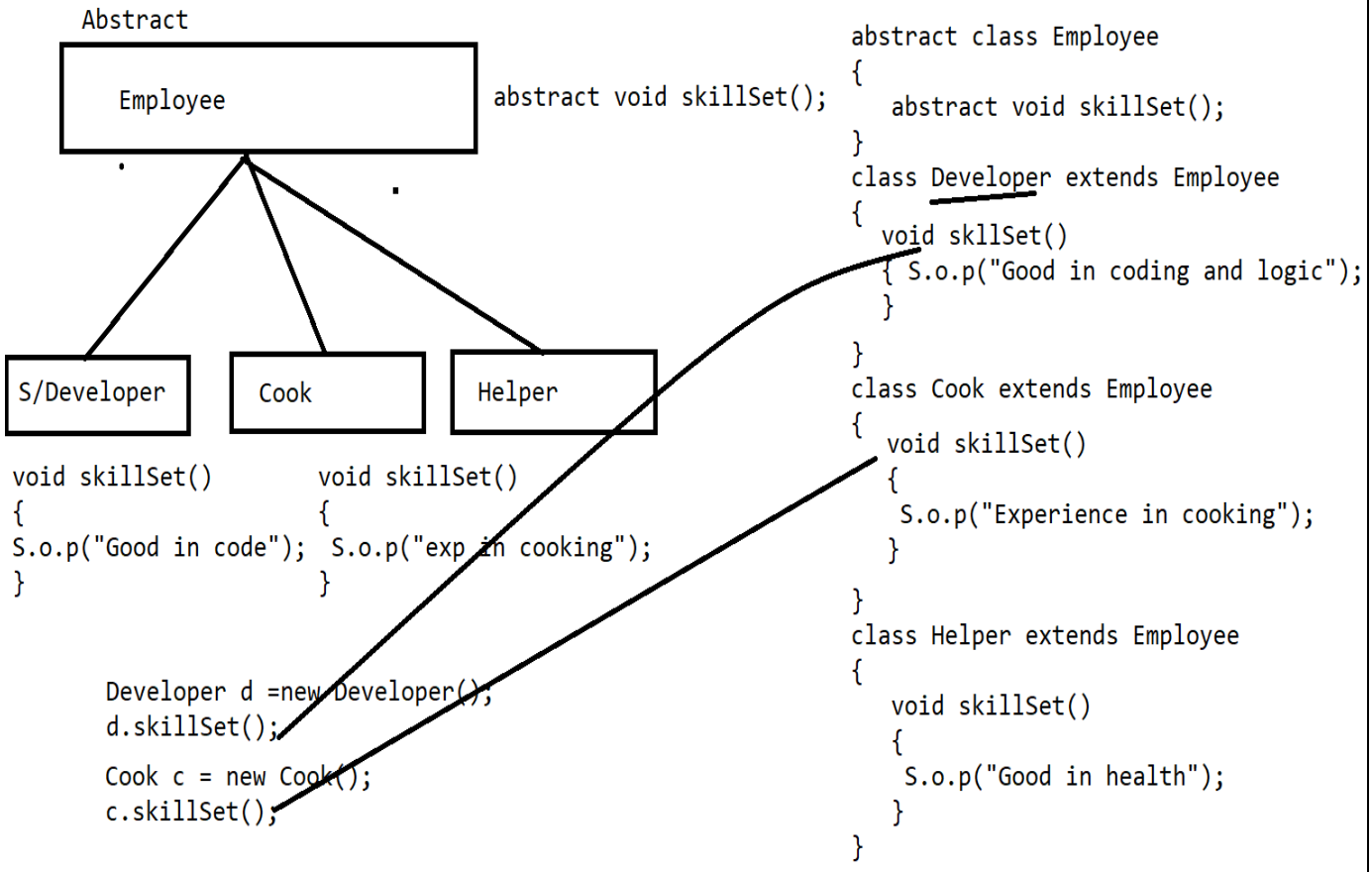
if we want to write the logic of abstract method we need to inherit the abstract class in any another class and override the abstract method and write its logic.

Q. What is the benefit of overriding abstract method in child class?

The benefit is we can modify the or we can write the different logic in every abstract class according to its requirement

Example

Suppose we want to hire the employee for different field but we required skillSet() for every employee but we cannot predict the skillSet() of employee. It is depend on in which field we want to hire employee.



Note: in the case of abstract class we not need to create the object of parent class. We can access the abstract member using its child class.

Following code show example of abstract class

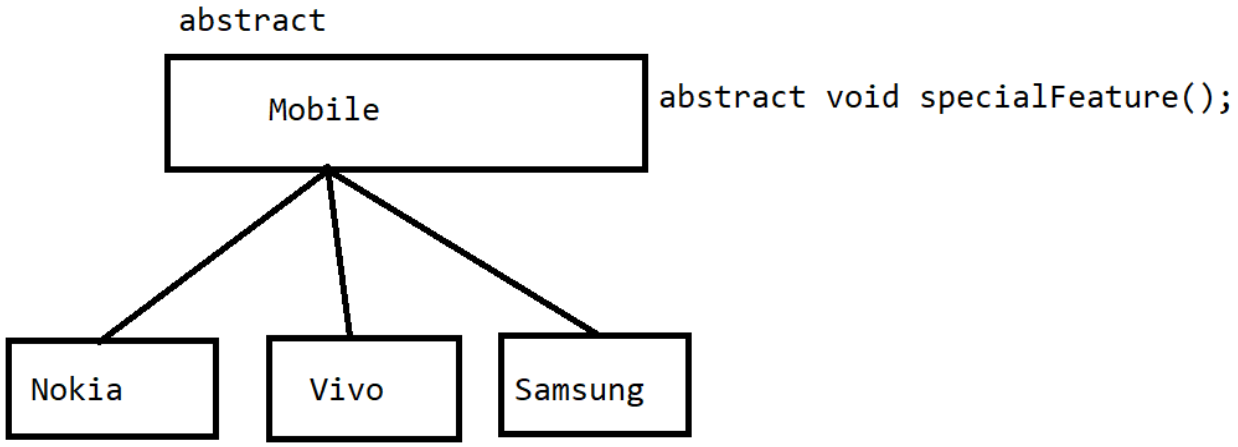
```
package org.techhub;
abstract class Employee {
	abstract void skillSet();
}
class Developer extends Employee {

	@Override
	void skillSet() {
		// TODO Auto-generated method stub
		System.out.println("Good in coding");
	}
}
class Cook extends Employee {
	@Override
	void skillSet() {
		// TODO Auto-generated method stub
		System.out.println("Exp in cooking");
	}
}
class Helper extends Employee
 {
	@Override
	void skillSet() {
		// TODO Auto-generated method stub
		System.out.println("Good In Health");
```

```
	}
}
public class AbstractApplication {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Developer d = new Developer();
		d.skillSet();
		Cook c = new Cook();
		c.skillSet();
		Helper h = new Helper();
		h.skillSet();
	}
}
```

One More Example of Abstraction

Suppose we want to purchase with some specialFeature()
Then we cannot predict the specialFeature() of mobile it is
Vary from company to company
So here we need to declare the one abstract class name as Mobile
With abstract method name as specialFeature () and we need to create the child classes of the mobile like as Nokia,Vivo etc
and we can write the different logic of specialFeature in every mobile child class.


abstract
Mobile
abstract void specialFeature();
Nokia
Vivo
Samsung

```
abstract class Mobile
{
  abstract void specialFeature();
}
class Nokia extends Mobile
{
   void specialFeature()
   { S.o.p("Good Battery Backup");
   }
}
class Vivo extends Mobile
{
   void specialFeature()
   { S.o.p("Good Camera Quality");
   }
}
class Samsung extends Mobile
{
   void specialFeature()
   { S.o.p("good in look");
   }
}
```

Example

```
package org.techhub;
abstract class Mobile
{
	abstract void specialFeature();
}
class Nokia extends Mobile
{
	@Override
```

```
        void specialFeature() {
                // TODO Auto-generated method stub
                System.out.println("Good In Battery Backup");
        }
}
class Vivo extends Mobile
{
        @Override
        void specialFeature() {
                // TODO Auto-generated method stub
                System.out.println("Good In Camera Quality");
        }

}
public class MobileExampleWithAbstraction {

        public static void main(String[] args) {
                // TODO Auto-generated method stub
    Nokia n = new Nokia();
      n.specialFeature();
    Vivo v = new Vivo();
      v.specialFeature();
       }

}
```

If we want to work with Abstract method and abstract class we have the some important points

---

i) Abstract class cannot create its object
ii) abstract method cannot have logic

iii) If we have abstract method in class then class must be abstract

Example:

```
class ABC
{  abstract void show ();
}
```

Above code is not valid because we cannot declare the abstract method in non abstract class. So if we want to declare the abstract method we have the correct given below

Example:

```
abstract class ABC
 {
     abstract void show();
  }
```

Note: in abstract class there is possibility non abstract method may be exist

```
abstract class ABC{
   abstract void show ();
        void display ()
        {
        }
 }
```

If we declare the abstract and non abstract method in abstract class Then class called as concrete class

iv) If abstract class contains more than one abstract methods then all method must be Override where abstract class get inherit.

```
abstract class A
      {
         abstract void show();
        abstract void display();
      }
class B extends A
  {
//we have error because we not override display()
//because A contain two abstract methods so we need to override
//all methods

          void show()   //override all method if we not required
              {
                System.out.println("I required show");
              }
      }
class C extends A
    {
//error we have error because we not override show()
  //because A contain two abstract methods so we need to override
//all methods
          void display()  //override all method if we not required.
```

```
        {
          System.out.println("I required display method");
        }
    }
```

If we want to solve above problem we must be override all abstract method in child class where we inherit the abstract class

```
abstract class A
{   abstract void show();
abstract void display();
}
class B extends A
{
void show()
{
System.out.println("I required show");
}
void display()
{
}
}
class C extends A
{
void display()
{
System.out.println("I required display method");
}
```

```
void show() {
}
}
```

Example

```
package org.techhub;

abstract class A
{ abstract void show();
  abstract void display();
}
class B extends A
{
	void show()
	{
		System.out.println("I required show method");
	}

	@Override
	void display() {
		// TODO Auto-generated method stub
	}
}
class C extends A
{
	void display()
	{
		System.out.println("I reuqired display method");
	}

	@Override
	void show() {
		// TODO Auto-generated method stub
```

```
	}
}
public class AbsMethodRuleApp {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
	 B b1 = new B();
	 b1.show();
	 C c1 = new C();
	 c1.display();
	}
}
```

Note: above approach is not good approach in coding

So if we want to above problem we have the one more approach called as adapter class.

Q. What is the Adapter class?

Adapter class is intermediator class which contain the all blank method definition of abstract class and which is able to provide the specific method to its child class called as adapter class.

```
package org.techhub;
abstract class ABC
{
	abstract void s1();
	abstract void s2();
	abstract void s3();
	abstract void s4();
	abstract void s5();
}
class ADP extends ABC
```

```
{       @Override
        void s1() {
                // TODO Auto-generated method stub
        }
        @Override
        void s2() {
                // TODO Auto-generated method stub
        }
        @Override
        void s3() {
        // TODO Auto-generated method stub
        }
        @Override
        void s4() {
                // TODO Auto-generated method stub
        }
        @Override
        void s5() {       // TODO Auto-generated method stub
        }
}
class FChild extends ADP
{
        void s1()
        {
                System.out.println("I need s1 method");
        }
}
class SChild extends ADP
{
         void s2()
         {
                 System.out.println("I need s2 method");
```

```
	}
}
public class AdapterImplementationApp
{
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		FChild fc= new FChild();
		fc.s1();
		SChild sc= new SChild();
		sc.s2();
	}
}
```

v) We cannot create object of abstract class but can create its reference

If we want to create reference of abstract class we need to create the object of its child class.

Example

```
package org.techhub;
abstract class Test
{ abstract void show();
}
class TestChild extends Test
{  @Override
	void show() {
		// TODO Auto-generated method stub
		System.out.println("I am abstract method");
	}
}
public class AbsRefApp {
	public static void main(String[] args) {
```

```
            // TODO Auto-generated method stub
    Test   t  = new TestChild();
    t.show();
      }

}
```

If we create the reference of abstract class using that reference we can call only those member declared within parent. We cannot access the any original member of child class using abstract class reference.

```
package org.techhub;
abstract class Test
{ abstract void show();
}
class TestChild extends Test
{  @Override
      void show() {
            // TODO Auto-generated method stub
            System.out.println("I am abstract method");
      }
  void display()
  {
        System.out.println("I am display method");
  }
}
public class AbsRefApp {
      public static void main(String[] args) {
            // TODO Auto-generated method stub
    Test   t  = new TestChild();
    t.show();
    t.display();
      }
```

}

Note: above code generate the compile time error to us The method display() is undefined for the type Test Because display() is original method of TestChild class and we try to call it using Test i.e. abstract class reference so it is not possible to call child class method using parent reference so compiler generate the error to us

But if we create the object of child and reference of child class then using that reference we can call the parent member as well as child member.

Example

```
package org.techhub;
abstract class Test
{ abstract void show();
}
class TestChild extends Test
{  @Override
        void show() {
                // TODO Auto-generated method stub
                System.out.println("I am abstract method");
        }
  void display()
  {
          System.out.println("I am display method");
  }
}
public class AbsRefApp {
        public static void main(String[] args) {
                // TODO Auto-generated method stub
     TestChild   t  = new TestChild();
     t.show();
```

```
    t.display();
      }

}
```

Note: What is the benefit of parent reference?

The Major benefit of parent reference is to achieve loose coupling.

Q .what is the meaning of coupling?

Coupling means if we one class is dependent on another class called as coupling.

There are two types of coupling?

1) **Tight Coupling**: tight coupling means if one class is 100% dependent on another class called as tight coupling**.**

2) **Loose Coupling**: loose coupling means if one class is partially dependent on another class called as loose coupling.

Example

```
package org.techhub.loosecoupling;
import java.util.*;
abstract class Mobile
{
      abstract void specialFeature();
}
class Nokia extends Mobile
{
      @Override
      void specialFeature() {
            // TODO Auto-generated method stub
            System.out.println("Good Battery backup");
```

```
	}

}
class Vivo extends Mobile
{
	@Override
	void specialFeature() {
		// TODO Auto-generated method stub
	 System.out.println("Good Camera Quality");
	}

}
class RamShowRoom
{
	void saleMobile(Mobile mobile)
	{
		mobile.specialFeature();
	}
}
public class LooseCouplingApp {

	 public static void main(String[] args) {
		// TODO Auto-generated method stub
	RamShowRoom r = new RamShowRoom();
	Scanner xyz  = new Scanner(System.in);
	System.out.println("1:Nokia");
	System.out.println("2:Vivo");
	System.out.println("Enter your choice");
	int choice=xyz.nextInt();
	switch(choice)
	{
	case 1:
```

```
        Mobile m=new Nokia();
         r.saleMobile(m);
         break;
       case 2:
         m=new Vivo();
         r.saleMobile(m);
         break;
        default:
              System.out.println("wrong choice");
      }
       }

}
```

Example

```
package org.techhub.loosecoupling;
import java.util.*;
abstract class Value
{  int first,second;
      abstract void setValue(int x,int y);
      abstract void performOperation();
}
class Add extends Value
{   @Override
      void setValue(int x, int y) {
            // TODO Auto-generated method stub
            first=x;
            second=y;
      }
      @Override
      void performOperation() {
            // TODO Auto-generated method stub
            System.out.printf("Addition is %d\n",first+second);
```

```
	}
}
class Mul extends Value
{  @Override
	void setValue(int x, int y) {
		// TODO Auto-generated method stub
		first=x;
		second=y;
	}
   @Override
	void performOperation() {
		// TODO Auto-generated method stub
		 System.out.printf("Multiplication is %d\n", first*second);
	}
}
class Sub extends Value
{	@Override
	void setValue(int x, int y) {
		 first=x;
		 second=y;
	}
	@Override
	void performOperation() {
		// TODO Auto-generated method stub
		System.out.printf("Substraction is %d\n",first-second);
	}
}
class Calculator
{
	void performCalculation(Value value)
	{
		value.setValue(10, 20);
```

```
                value.performOperation();
        }
}
public class CalculateApplication {
    public static void main(String[] args) {
                // TODO Auto-generated method stub
    Scanner xyz = new Scanner(System.in);
     Calculator c = new Calculator();
       int choice;
     Value v=null;
      System.out.println("Enter your choice");
      choice=xyz.nextInt();
      switch(choice)
      {
      case 1:
           v=new Add();
           c.performCalculation (v);
           break;
      case 2:
            v =new Mul();
            c.performCalculation (v);
           break;
      case 3:
           v=new Sub();
           c.performCalculation(v);
           break;
           default :
                  System.out.println ("Wrong choice");
      }
        }

}
```