Polymorphism in Java

- Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.
- If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism or Dynamic Method Dispatch in Java

- Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.
- Let's See some examples to understand runtime polymorphism

Example:1

```
class A
{
        void m1()
        {
                System.out.println("Parent");
        }
}
class B extends A
{
        void m1()
        {
                System.out.println("Child 1");
        }
}
class C extends A
{
        void m1()
        {
                System.out.println("Child 2");
        }
}
class Main
{
        public static void main(String [] args)
        {
                A a=new A();
```

```
                B b=new B();
                C c=new C();
                A ref;
                ref=a;
                ref.m1();
                ref=b;
                ref.m1();
                ref=c;
                ref.m1();
        }
}
```

Output

E:\JAVA-2>java Main
Parent
Child 1
Child 2

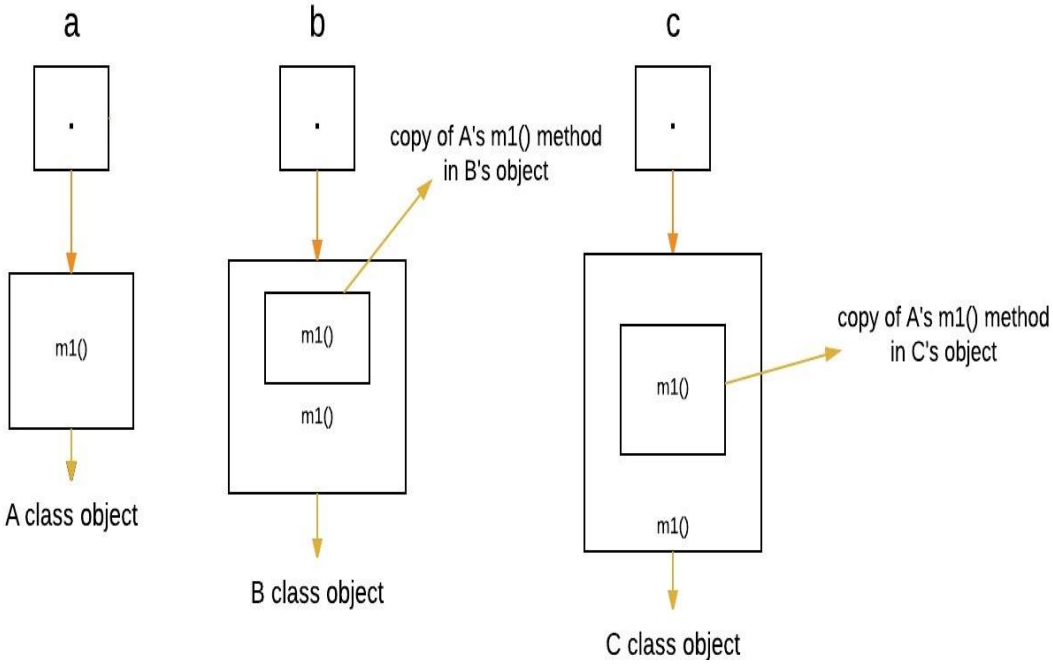
- **The above program creates one superclass called A and it's two subclasses B and C. These subclasses overrides m1( ) method.**

(1) Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.

A a = new A(); // object of type A

B b = new B(); // object of type B

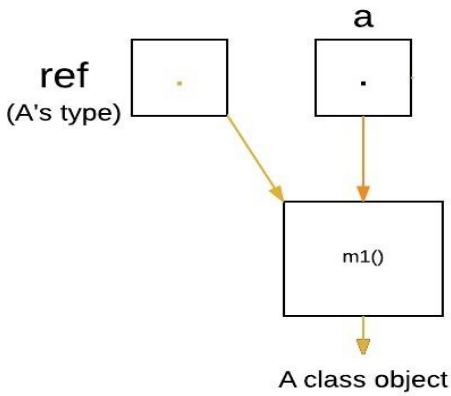
C c = new C(); // object of type C



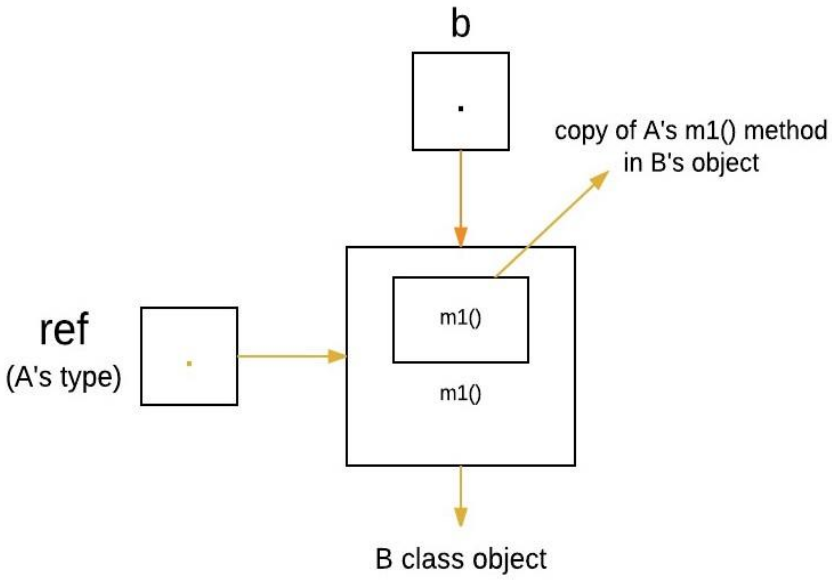
(2) Now a reference of type A, called ref, is also declared, initially it will point to null.
A ref; // obtain a reference of type A

(3) Now we are assigning a reference to each type of object (either A's or B's or C's) to ref, one-by-one, and uses that reference to invoke m1( ). As the output shows, the version of m1( ) executed is determined by the type of object being referred to at the time of the call.

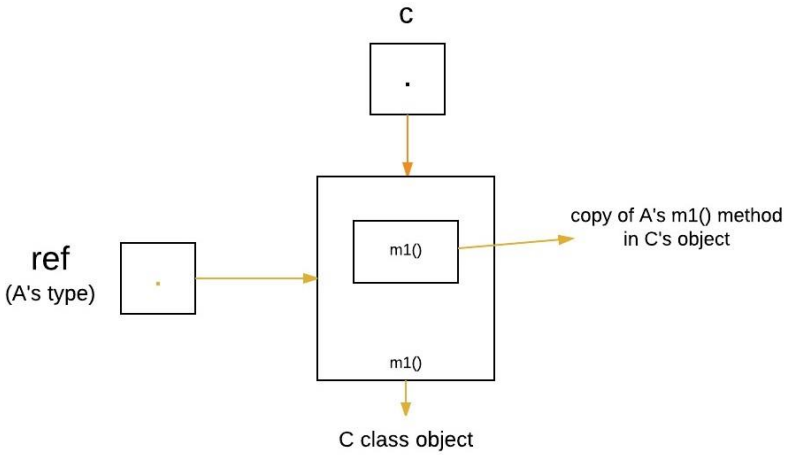
ref = a; // ref refers to an A object
ref.m1(); // calling A's version of m1()



ref = b; // now ref refers to a B object
ref.m1(); // calling B's version of m1()



ref = c; // now ref refers to a C object
ref.m1(); // calling C's version of m1()



Example:2

```
class A
{
        void m1()
        {
                System.out.println("Parent");
        }
}
class B extends A
{
        void m1()
        {
                System.out.println("Child 1");
        }
}
class C extends A
{
        void m1()
        {
                System.out.println("Child 2");
        }
}
class Main
{
        public static void main(String [] args)
        {
                A a=new A();
```

```
                A a1=new B();
                A a2=new C();
                a.m1();
                a1.m1();
                a2.m1();
        }
}
```

Output

```
E:\JAVA-2>java Main
Parent
Child 1
Child 2
```

Runtime Polymorphism with Data Members

- In Java, we can override methods only, not the variables(data members), so runtime polymorphism cannot be achieved by data members.
- For example :

```
class A
{
        int x=10;
        void get()
        {
                System.out.println(x);
        }
}
class B extends A
{
        int x=20;
        void get()
        {
                System.out.println(x);
        }
}
class C extends A
{
        int x=30;
        void get()
        {
                System.out.println(x);
        }
}
class Main
{
        public static void main(String [] args)
        {
```

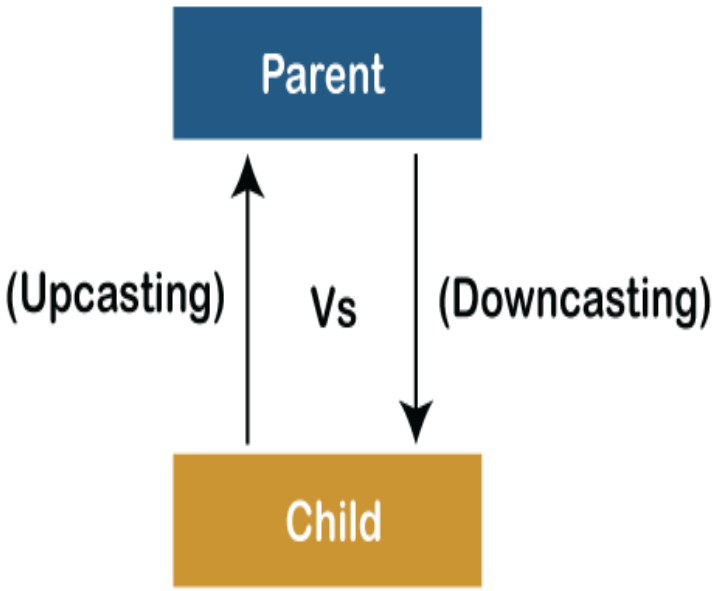
```
                A a=new A();
                B b=new B();
                C c=new C();
                System.out.println(a.x);
                a=b;
                System.out.println(a.x);
                a=c;
                System.out.println(a.x)
        }
}
```

Output

```
E:\JAVA-2>java Main
10
10
10
```

Upcasting and Downcasting in Java

- A process of converting one data type to another is known as Typecasting and Upcasting and Downcasting is the type of object typecasting.
- In Java, the object can also be typecasted like the datatypes. Parent and Child objects are two types of objects.
- There are two types of typecasting possible for an object, i.e., **Parent to Child** and **Child to Parent** or can say **Upcasting and Downcasting.**
- Typecasting is used to ensure whether variables are correctly processed by a function or not.
- In Upcasting and Downcasting, we typecast a child object to a parent object and a parent object to a child object simultaneously.
- We can perform Upcasting implicitly or explicitly, but downcasting cannot be implicitly possible.



(1) Upcasting

- Upcasting is a type of object typecasting in which a child object is typecasted to a parent class object.
- In upcasting we can use parent class reference object to refer the child class object.
- Here, we don't access all the variables and the method. We access only some specified variables and methods of the child class**. Upcasting** is also **known as Generalization** and **Widening.**

Example:1:- Explicit Upcasting

```
class  Parent
{
        void PrintData()
        {
                System.out.println("method of parent class");
        }
}
class Child extends Parent
{
        void PrintData()
        {
                System.out.println("method of child class");
        }
}
class UpcastingExample
{
        public static void main(String args[])
        {
                Parent obj1 = (Parent) new Child();//Explicit Type casting
                Parent obj2 = (Parent) new Child();
                obj1.PrintData();
                obj2.PrintData();
        }
}
```

Output

```
E:\JAVA-2>java UpcastingExample
method of child class
method of child class
```

Example:2:- Implicit Downcasting

```
 class  Parent
{
        void PrintData()
        {
                System.out.println("method of parent class");
```

```
        }
    }
    class Child extends Parent
    {
        void PrintData()
        {
            System.out.println("method of child class");
        }
    }
    class UpcastingExample
    {
        public static void main(String args[])
        {
            Parent obj1 =new Child();//Implicit Downcasting
            Parent obj2 =new Child();
            obj1.PrintData();
            obj2.PrintData();
        }
    }
```

Output

```
E:\JAVA-2>java UpcastingExample
method of child class
method of child class
```

Example:3:-

```
class A
{
    int x=10;
    void get()
    {
        System.out.println(x);
    }
}
class B extends A
{
    int x=20;
    void get()
    {
        System.out.println(x);
    }
    void print()
    {
        System.out.println(x);
    }
}
```

```
class C extends A
{
        int x=30;
        void get()
        {
                System.out.println(x);
        }
}
class Main
{
        public static void main(String [] args)
        {
                A a=new A();
                B b=new B();
                C c=new C();
                a.get();
                A a1=b; //upcasting
                a1.get();
                A a2=c; //upcasting
                a2.get();
        }
}
```

Output

```
E:\JAVA-2>java Main
10
20
30
```

(2) Downcasting

- As we know,In Upcasting, we assign a parent class reference object to the child class. In Java, we cannot assign a parent class reference object to the child class.
- but if we perform downcasting, we will not get any compile-time error.
- However, when we run it, it throws the "ClassCastException". Now the point is if downcasting is not possible in Java, then why is it allowed by the compiler? In Java, some scenarios allow us to perform downcasting. Here, the subclass object is referred by the parent class**.**

Example:1:-

```
class A
{
        int x=10;
        void get()
        {
                System.out.println(x);
        }
}
```

```
class B extends A
{
        int x=20;
        void get()
        {
                System.out.println(x);
        }
}
class Main
{
        public static void main(String [] args)
        {
                //A a=new A();//Runtime Error
              /*Exception in thread "main" java.lang.ClassCastException:
              class A cannot be cast to class B*/
               A a=new B();
                B b=new B();
                b=(B)a;
                b.get();
                System.out.println(a.x);
        }
}
```

Output

```
E:\JAVA-2>java Main
20
10
```

Example:2:-

```
class Parent
{
        String name;
        void showMessage()
        {
                System.out.println("Parent method is called");
        }
}
class Child extends Parent
        {
        int age;
        void showMessage()
        {
                System.out.println("Child method is called");
        }
}
```

```
class Downcasting
{
	public static void main(String[] args)
	{
		Parent p = new Child();
		p.name = "Shubham";

		// Performing Downcasting Implicitly
		//Child c = new Parent(); // it gives compile-time error

		// Performing Downcasting Explicitly
		Child c = (Child)p;

		c.age = 18;
		System.out.println(c.name);
		System.out.println(c.age);
		c.showMessage();
	}
}
```

Output

```
E:\JAVA-2>java Downcasting
Shubham
18
Child method is called
```

Java instanceof operator

- The java **instanceof** operator is used to test whether the object is an instance of the specified type (class or subclass or interface).
- The **instanceof** in java is also known as type comparison operator because it compares the instance with type. It returns either true or false.
- If we apply the **instanceof** operator with any variable that has **null** value, it returns false.

Example:1:-

```
class Student
{
	int x=10;
	void get()
	{
		System.out.println(x);
	}
}
class LjStudent extends Student
{
	int x=20;
	void get()
```

```
        {
                System.out.println(x);
        }
}
class Main
{
        public static void main(String [] args)
        {
                Student S1=new Student();
                System.out.println("S1 is object of student:"+(S1 instanceof Student));
                Student S2=new LjStudent();
                System.out.println("S2 is object of student:"+(S2 instanceof Student));
                System.out.println("S2 is object of Ljstudent:"+(S2 instanceof
                                LjStudent));
                System.out.println("S1 is object of Ljstudent:"+(S1 instanceof
                                 LjStudent));
                Student S3=null;
                System.out.println("S3 is object of student:"+(S3 instanceof Student));
        }
}
```

Output

```
E:\JAVA-2>java Main
S1 is object of student:true
S2 is object of student:true
S2 is object of Ljstudent:true
S1 is object of Ljstudent:false
S3 is object of student:false
```

Example:2:Real use of instanceof operator

```
class Student
{
        void get()
        {
                System.out.println("x");
        }
}
class LjStudent extends Student
{
        void checkObject(Student s)
        {
                if(s instanceof LjStudent)
                {
                        LjStudent lj=(LjStudent)s;// Down casting
                        System.out.println("Down Casting Of Lj student Sucess");
                }
```

```
                else
                {
                        System.out.println("Down Casting Of Lj student NotSucess");
                }
        }
}
class LdStudent extends Student
{
        void checkObject(Student s)
        {
                if(s instanceof LdStudent)
                {
                        LdStudent ld=(LdStudent)s;
                        System.out.println("Down Casting Of Ld student Sucess");
                }
                else
                {
                        System.out.println("Down Casting Of Ld student NotSucess");
                }
        }
}
class Main
{
        public static void main(String [] args)
        {
                LjStudent lj=new LjStudent();
                LdStudent ld=new LdStudent();
                Student s;
                s=lj;
                lj.checkObject(s);
                s=ld;
                ld.checkObject(s);
                lj.checkObject(s);
        }
}
```

Output

```
E:\JAVA-2>java Main
Down Casting Of Lj student Sucess
Down Casting Of Ld student Sucess
Down Casting Of Lj student NotSucess
```

Abstraction

- Data abstraction is the process of hiding certain details and showing only essential information to the user.
- Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.
- Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.
- Abstraction can be achieved with either abstract classes or interfaces.

Java abstract Keyword

- The abstract keyword is used to achieve abstraction in Java. It is a non-access modifier which is used to create abstract class and method.
- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from 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).

Rules of abstract keyword

❖ **Don'ts**

- An abstract keyword cannot be used with variables and constructors.
- If a class is abstract, it cannot be instantiated.
- If a method is abstract, it doesn't contain the body.
- We cannot use the abstract keyword with the final.
- We cannot declare abstract methods as private.
- We cannot declare abstract methods as static.
- An abstract method can't be synchronized.

❖ **Do's**

- An abstract keyword can only be used with class and method.
- An abstract class can contain constructors and static methods.
- If a class extends the abstract class, it must also implement at least one of the abstract method.
- An abstract class can contain the main method and the final method.
- An abstract class can contain overloaded abstract methods.
- We can declare the local inner class as abstract.
- We can declare the abstract method with a throw clause.

Java Abstract Class

- The abstract class in Java cannot be instantiated **(we cannot create objects of abstract classes)**. We use the abstract keyword to declare an abstract class.

Example:1:- Cannot Create object of abstract class

```
abstract class Shape
{
}
class Main
{
        public static void main(String [] args)
        {
                /*Shape s; Can create reference */
                Shape s=new Shape();//Compiler Error
        }
}
/*E:\JAVA-2>javac Ex5.java
Ex5.java:8: error: Shape is abstract; cannot be instantiated
            Shape s=new Shape();*/
```

Example:2:- Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass.

```
abstract class Language
{
        // method of abstract class
        void display()
        {
                System.out.println("This is Programming Language");
        }
}

class Java extends Language
{
        void show()
        {
                System.out.println("Java is Programming Language");
        }
}
class Main
{
        public static void main(String[] args)
        {
                Language l=new Java();
                l.display();
                //l.show();
                /*Ex5.java:40: error: cannot find symbol
              l.show();
```

```
                ^
                    symbol:   method show()
                    location: variable l of type Language*/
                    Java j=new Java();
                    j.display();
                    j.show();
            }
    }
```

Output

```
E:\JAVA-2>java Main
This is Programming Language
This is Programming Language
Java is Programming Language
```

Java Abstract Method

- A method that doesn't have its body or does not have implementation is known as an abstract method. We use the same abstract keyword to create abstract methods.

Example:3:-If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error.

```
class Language
{
        // method of abstract class
        abstract void display();
//Here, display() is an abstract method. The body of display() is replaced by ;.
}

class Main
{
        public static void main(String[] args)
        {

        }
}
```

Output

Ex5.java:51: error: Language is not abstract and does not override abstract method display() in Language

class Language

```
abstract class Language
{
        // method of abstract class
        abstract void display();

}

class Main
{
        public static void main(String[] args)
        {

        }
}
```

***No compiler error**

Conclusion:

- **if you have at least one method abstract then the class must be abstract**

Example:4:-Implementation of abstract Method

```
abstract class Shape
{
	abstract void display();//Abstract method does not have any implementation
	abstract void area(int a);
	void show()//Regular Method
	{
		System.out.println("Hello");
	}
}
class Rectangle extends Shape
{
	void display()// implementation abstract method
	{
		System.out.println("Rectangle");
	}
	void area(int b)// implementation abstract method
	{
		System.out.println("Rectangle area");
	}
}
class Circle extends Shape
{
	void display()// implementation abstract method
	{
		System.out.println("Circle");
	}
	void area(int b)// implementation abstract method
	{
		System.out.println("Circle area");
	}
}
class Main
{
	public static void main(String [] args)
	{
		Shape s=new Rectangle();
		s.display();
		s.area(10);
		s.show();
		Shape s1=new Circle();
		s1.display();
		s1.area(10);
		s1.show();
	}
}
```

Conclusions:

- **All Abstract Method Must be implemented in Child class.**
- **If you do not want to implement abstract method in child class then make the child class abstract.**
- **Abstract method can't be final, static, private and synchronized.**

Example:5:- Access no args Constructor of abstract parent class

```
abstract class Shape
{
	Shape()
	{
		System.out.println("Constructor Parent");
	}
	abstract void display();//Abstract method does not have any implementation
	abstract void area(int a);
	void show()// implementation abstract method
	{
		System.out.println("Hello");
	}
	static void print()// Static Method
	{
		System.out.println("Hello Static");
	}
}
class Rectangle extends Shape
{
	void display()// implementation abstract method
	{
		System.out.println("Rectangle");
	}
	void area(int b)// implementation abstract method
	{
		System.out.println("Rectangle area");
	}
}
class Main
{
	public static void main(String [] args)
	{
		Shape s=new Rectangle();
		s.display();
		s.area(10);
		s.show();
		Shape.print();
	}
}
```

Output:
E:\JAVA-2>java Main
Constructor Parent
Rectangle
Rectangle area
Hello
Hello Static

Example:6:-Access Parametric Constructor of abstract parent class

```
abstract class Shape
{
    Shape(int a)
    {
        System.out.println("Parametric Constructor Parent");
    }
    abstract void display();//Abstract method does not have any implementation
    abstract void area(int a);
    void show()// implementation abstract method
    {
        System.out.println("Hello");
    }
    static void print()// Static Method
    {
        System.out.println("Hello Static");
    }

}
class Rectangle extends Shape
{
    Rectangle()
    {
        super(30);//Parametric constructor of abstract parent class called
    }
    void display()// implementation abstract method
    {
        System.out.println("Rectangle");
    }
    void area(int b)// implementation abstract method
    {
        System.out.println("Rectangle area");
    }
}
class Main
{
    public static void main(String [] args)
```

```
        {
                Shape s=new Rectangle();
                s.display();
                s.area(10);
                s.show();
                Shape.print();

        }
    }
```

Output:

```
E:\JAVA-2>java Main
Parametric Constructor Parent
Rectangle
Rectangle area
Hello
Hello Static
```

Example:6:-Access abstract inner class of abstract parent class

```
    abstract class Car
    {
            abstract class Engine
            {
                    abstract void ignition();
                    void engineType()
                    {
                            System.out.println("Petrol Engine");
                    }
            }

    }
    class Test1 extends Car
    {
            class Test2 extends Engine
            {
                    void ignition()
                    {
                            System.out.println("Spark Ignition");
                    }
            }
    }
    class Main
    {
            public static void main(String [] args)
            {
                    Test1 t1=new Test1();
                    Test1.Test2 t2=t1.new Test2();
```

```
                t2.engineType();
                t2.ignition();
        }
}
```

Output:

```
E:\JAVA-2>java Main
Petrol Engine
Spark Ignition
```

Conclusion:

- **The inner class of child class must extends the inner abstract class of parent**