Note:

Java Enum internally inherits the Enum class, so it cannot inherit any other class, but it can **implement** many **interfaces**. We can have fields, **constructors**, **methods**, and **main methods** in Java enum.

Points to remember for Java Enum

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods

Examples:

Simple Example of Java Enum

```
class EnumExample1{

public enum Season { WINTER, SPRING, SUMMER, FALL }

public static void main(String[] args)
{

//traversing the enum

for (Season s : Season.values())

System.out.println(s);

}}
```

```
class EnumExample1{
public enum Season { WINTER, SPRING, SUMMER, FALL }
public static void main(String[] args) {
for (Season s : Season.values()){
System.out.println(s);
}
System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());
}
}
```

Output

```
WINTER

SPRING

SUMMER

FALL
```

Output

```
WINTER
SPRING
SUMMER
FALL
Value of WINTER is: WINTER
Index of WINTER is: 0
Index of SUMMER is: 2
```

Note: Java compiler internally adds **values(), valueOf() and ordinal()** methods within the enum at compile time. It internally creates a **static and final** class for the enum.

- What is the purpose of the values() method in the enum?

A: The values () method returns an array containing all the values of the enum.

-What is the purpose of the valueOf() method in the enum?

A: The valueOf() method returns the value of given constant enum.

-What is the purpose of the ordinal() method in the enum?

A: The ordinal () method returns the index of the enum value.

Defining Java Enum

Note : The enum can be defined **within** or **outside** the class because it is similar to a class.

Initializing specific values to the enum constants

The enum constants have an initial value which starts from **0, 1, 2, 3**, and so on. But, we can initialize the specific value to the enum constants by defining fields and constructors. As specified earlier, Enum can have fields, constructors, and methods.

```
class EnumExample4{
enum Season{
WINTER(5), SPRING(10), SUMMER(15), FALL(20);
private int value;
private Season(int value){
this.value=value;
}
}
public static void main(String args[]){
for (Season s : Season.values())
System.out.println(s+" "+s.value);
 }
```

Output

```
WINTER 5

SPRING 10

SUMMER 15

FALL 20
```

}

Note : constructor of enum type is private. If you don't declare private compiler internally creates private constructor.

```
enum Season{
WINTER(10),SUMMER(20);
private int value;
Season(int value){
this.value=value;
}
}
```

Can we create the instance of Enum by new keyword?

A:No, because it contains private constructors only.

Can we have an abstract method in the Enum?

A:Yes, Of course! we can have abstract methods and can provide the implementation of these methods.

```
enum Size {
  // enum constants calling the enum constructors
  SMALL("The size is small."),
  MEDIUM("The size is medium."),
  LARGE("The size is large."),
  EXTRALARGE("The size is extra large.");

private final String pizzaSize;

  // private enum constructor
  private Size(String pizzaSize) {
    this.pizzaSize = pizzaSize;
  }

  public String getSize() {
    return pizzaSize;
  }
}
```

```
class Main {
  public static void main(String[] args) {
    Size size = Size.SMALL;
    System.out.println(size.getSize());
  }
}
```

Output
The size is small.

Polymorphism

The word polymorphism means having **many forms**. In simple words, we can define polymorphism **as the ability of a message to be displayed in more than** one form.

What is Polymorphism?

The term polymorphism means that an object can exist in different crystalline forms. For example, carbon can exist in three common types. Coal, graphite, and diamond are the three different crystalline forms of carbon.

Similarly, in Java, Polymorphism is a phenomenon of an object that can exhibit a property of performing mathematical and logical operations from different perspectives.

Real-life Illustration: Polymorphism

A person at the same time can have different characteristics. **Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.**

- Polymorphism is considered one of the important features of Object-Oriented Programming.

- Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations.
- The word “poly” means many and “morphs” means forms, So it means many forms.

Why Polymorphism?
To reducing complexity,

Java example

Where the polymorphism can only be achieved through the behavior (methods), the method "**println**()", single name many form. It is an example of polymorphism.
- println(10);
- println("HelloJava");
- println(23.4);
Hence you can see only one method name to print all the different values instead of having different functions to print the values of different data types.

Run-time polymorphism comes in three different forms:
1. use **up casting**, this the most common use of polymorphism in OOP.
2. run-time polymorphism with **abstract** base classes
3. Run-time polymorphism with **interfaces**.

Example:

```
class one{
void print(){
System.out.println("In class one");}
}
class two extends one{
void print(){
System.out.println("In class two");}
public static void main(String args[]){
one a = new two(); //upcasting
```

Out put

In class two

```
a.print(); }
}
```

Types of polymorphism

In Java polymorphism is mainly divided into two types:

- **Compile-time Polymorphism**
- **Runtime Polymorphism**

Type 1: Compile-time polymorphism

It is also known as **static polymorphism**. This type of polymorphism is achieved by **function overloading**.

Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.

Example 1:

```
class Helper {
   static int Multiply(int a, int b)
   {
       return a * b;

   }
   static double Multiply(double a, double b)
   {
      return a * b;
   }
}
```

Output:

8

34.65

```
Class  test {

    public static void main(String[] args)
   {
      System.out.println(Helper.Multiply(2, 4));
      System.out.println(Helper.Multiply(5.5, 6.3));
```

```
    }
}
```

Example 2:

```
class Helper {
    static int Multiply(int a, int b)
    {
        return a * b;
    }
    static int Multiply(int a, int b, int c)
    {
        return a * b * c;
    }
}

class Test {

    public static void main(String[] args)
    {

        System.out.println(Helper.Multiply(2, 4));

        System.out.println(Helper.Multiply(2, 7, 3));

    }
}
```

Output:

8

42

Type 2: Runtime polymorphism

It is also known as **Dynamic** Method Dispatch. It is a process in which a function call to the **overridden** method is resolved at Runtime. This type of polymorphism is achieved by **Method Overriding**. Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

Example

```
class Parent {
   void Print()
   {
      System.out.println("parent class");
   }
}


class subclass1 extends Parent {

   void Print() { System.out.println("subclass1"); }

}

class subclass2 extends Parent {

      void Print()
   {
      System.out.println("subclass2");
   }
}
class Test {
      public static void main(String[] args)
   {
      Parent a;
      a = new subclass1();
      a.Print();
      a = new subclass2();
      a.Print();
   }
}
```

Output:

subclass1

subclass2

Output explanation:

Here in this program, when an object of child class is created, then the method inside the child class is called. This is because the method in the parent class is overridden by the child class. Since the method is overridden, this method has more priority than the parent method inside the child class. So, the body inside the child class is executed.

Example:

```
Public abstract class Shape {
    public abstract void shapeForm();
}
public class Square extends Shape {
    @Override
    public void shapeForm() {
        System.out.println("* * * *\n* * * *\n* * * *\n");
    }
}
public class Rectangle extends Shape {
    @Override
    public void shapeForm() {
        System.out.println("* * * * * *\n* * * * * *\n* * * * * *\n");
    }
}
public class Triangle extends Shape {
    @Override
```

```
    public void shapeForm() {
        System.out.println("    *\n   * *\n  * * *\n * * * *\n* * * * *\n");
    }
 }
public class Circle extends Shape {
     @Override
    public void shapeForm() {
        System.out.println("  * * *\n* * * * *\n* * * * *\n  * * *\n");
    }
 }
public class Drawer {
     public void draw(Shape s) {
        s.shapeForm();
    }
}
public class Main {
    public static void main(String[] args) {
         Shape s = new Square();
         Shape r = new Rectangle();
         Shape t = new Triangle();
         Shape c = new Circle();
         Drawer drawer = new Drawer();
        drawer.draw(s);
```

Output

```
* * * *
* * * *
* * * *

* * * * * *
* * * * * *
* * * * * *

    *
   * *
  * * *
 * * * *
* * * * *

  * * *
 * * * * *
 * * * * *
  * * *
```

```
        drawer.draw(r);
        drawer.draw(t);
        drawer.draw(c);
    }
}
```