Polymorphism

Polymorphism in Java — Edureka

In the real world, you might have seen a chameleon changing its color as per its requirement. If someone asks, “How it does that?”, you can simply say, “Because, it is polymorphic”. Similarly, in the programming world, Java objects possess the same functionality where each object can take multiple forms. This property is known as Polymorphism in Java, where Poly means many and morph means

change (or ‘form’). In this article, let’s discuss this key concept of Object Oriented Programming i.e. Polymorphism in Java.

Below are the topics to be covered in this article:

- What is Polymorphism?
- Polymorphism in Java with Example
- Types of Polymorphism in Java

1. Static Polymorphism
2. Dynamic Polymorphism

- Other Characteristics of Polymorphism in Java

What is Polymorphism?

Polymorphism is the ability of an entity to take several forms. In object-oriented programming, it refers to the ability of an object (or a reference to an object) to take different forms of objects. It allows a common data-gathering message to be sent to each class.

Polymorphism encourages called as ‘extendibility’ which means an object or a class can have it’s uses extended.

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***.

Now, let's understand this by taking a real-life example and see how this concept fits into Object-oriented programming.

Polymorphism in Java with Example

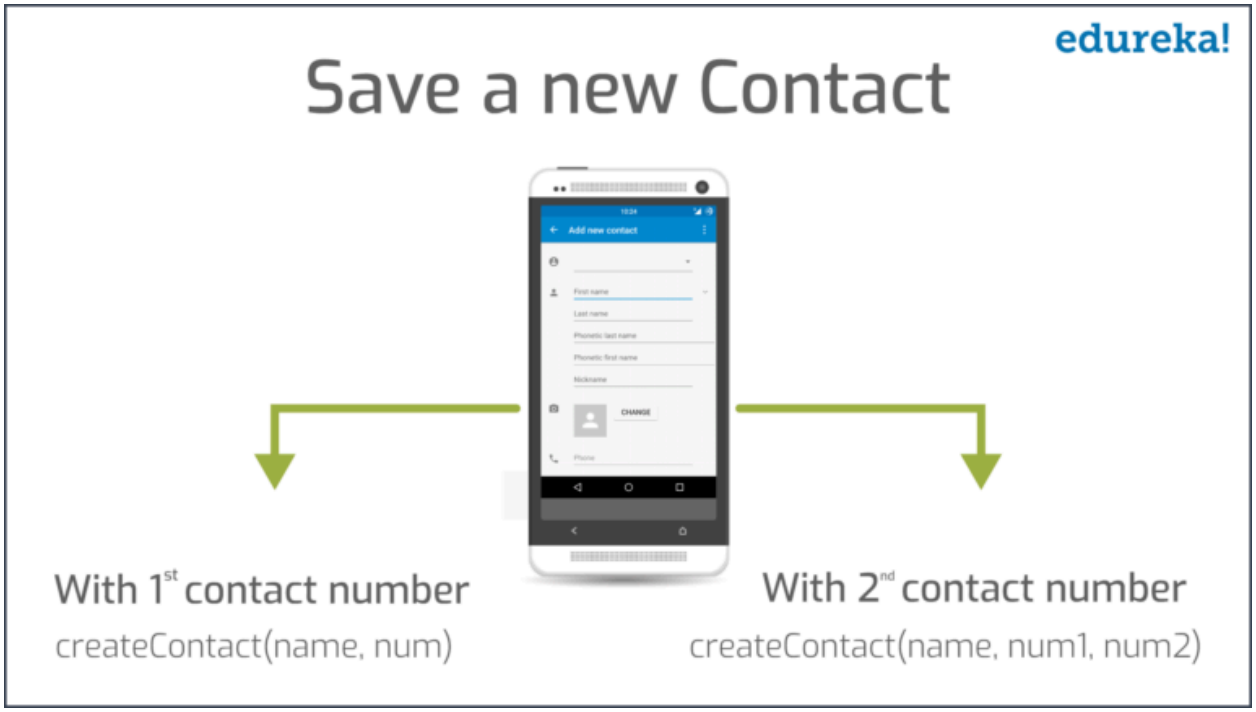
Let's understand this (the example) with the help of below problem statement.

Consider a cell phone where you save your Contacts. Suppose a person has two contact numbers. For the ease of accessibility, your cellphone provides you the functionality where you can save two numbers under the same name.

Similarly, in Java, an object is only one but it can take multiple forms depending on the context of the program. Suppose you want to write a

function to save two contact numbers of the same person, you can create it like — **void createContact(String name, int number1, int number2)**.

Now, it's not necessary that everyone in your contact list will have two contact numbers. Few of them might be having only a single contact number. In such situations, instead of creating another method with a different name to save one number for a contact, what you can do is, create another method with the same name i.e. **createContact().** But, instead of taking two contact numbers as parameters, take only one contact number as a parameter i.e. **void createContact(String name, int number1)**.



As you can see in the above figure, **createContact()** method has two different definitions. Here, which definition is to be executed depends upon the number of parameters being passed. If one parameter is passed, then only a single contact number is saved under the contact. But, if two contact numbers are passed to this method at the same time, then both will be saved under the same contact. *This is also known as* ***Method Overloading***.

Now let’s take another example and understand polymorphism in depth.

Suppose you went to a Shopping Centre (Allen Solly) near your home and bought a pair of jeans. A week later, while traveling to a nearby town, you spot another Shopping center. You walk into the shop and find a new variant of the same brand which you liked even more. But you decided to buy it from the shop near to your home. Once back home, you again went to the Shopping Center near your home to get those amazing pair of Jeans but couldn’t find it. Why? Because that was a specialty of the shop that was located in the neighboring town.

Now relating this concept to an object-oriented language like Java, suppose you have a class named XJeans which includes a method named **jeans()**. Using this method, you can get an Allen Solly jeans. For the Jeans in the neighboring town, there is another class **YJeans**. Both the classes XJeans and YJeans extends the parent class

ABCShoppingCenter. The YJeans class includes a method named **jeans()**, using which you can get both the jeans variants.

```
classABCShoppingCenter {

public void jeans() {

System.out.println("Default AllenSolly Jeans");

}

}

class XJeans extends ABCShoppingCenter {

public void jeans() {

System.out.println("Default AllenSolly Jeans");

}

}
```

```
class YJeans extends ABCShoppingCenter {

// This is overridden method

public void jeans() {

System.out.println("New variant of AllenSolly");

}

}
```

So, instead of creating different methods for every new variant, we can have a single method jeans**()**, which can be defined as per the different child classes. Thus, the method named **jeans()** has two definitions — one with only default jeans and other with both, the default jeans and the new variant. Now, which method gets invoked will depend on the type of object it belongs to. If you create **ABCShoppingCenter** class object, then there will be only one jeans available. But if you create **YJeans** class object, that extends **ABCShoppingCenter** class, then you can have both the variants. *This is also known as* ***Method***

Overriding. Thus, Polymorphism increases the simplicity and readability of the code by reducing the complexity. This makes Polymorphism in Java a very useful concept and it can be applied in real-world scenarios as well.

I hope you got an idea on the concept of Polymorphism. Now, let's move further with this article and understand different types of **Polymorphism in Java**.

Types of Polymorphism in Java

Java supports two types of polymorphism and they are as follows:

- Static Polymorphism
- Dynamic Polymorphism

Static Polymorphism

A polymorphism that is resolved during compile time is known as static polymorphism. Method overloading is an example of compile-time polymorphism.

Example

Method Overloading is a feature that allows a class to have two or more **method** to have the same name, but with different parameter lists. In the below example, you have two definitions of the same method add(). So, which add() method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

```
class Calculator

{

int add(int x, int y)
```

```
{

return x+y;

}

int add(int x, int y, int z)

{

return x+y+z;

}

}

public class Test

{

public static void main(String args[])

{
```

```
Calculator obj = new Calculator();

System.out.println(obj.add(100, 200));

System.out.println(obj.add(100, 200, 300));

}

}
```

This is how Static Polymorphism works. Now, let's understand what is Dynamic Polymorphism in Java.

Dynamic Polymorphism

Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, that's why it is called runtime polymorphism. Method Overriding is one of the ways to achieve Dynamic Polymorphism. In any object-oriented programming language, **Overriding** is a feature that allows a subclass or child class

to provide a specific implementation of a **method** that is already provided by one of its super-classes or parent classes.

Example

In the below example, you have two classes **MacBook** and **iPad**. *MacBook* is a parent class and *iPad* is a child class. The child class is overriding the method **myMethod()** of the parent class. Here, I have assigned child class object to the parent class reference to determine which method would be called at run-time. It is the type of object that determines which version of the method would be called (not the type of reference).

```
class MacBook{

public void myMethod(){

System.out.println("Overridden Method");

}
```

```
}

public class iPad extends MacBook{

public void myMethod(){

System.out.println("Overriding Method");

}

public static void main(String args[]){

MacBook obj = new iPad();

obj.myMethod();

}

}
```

Output:

Overriding Method

When you invoke the overriding method, then the object determines which method is to be executed. Thus, this decision is made at run time.

I have listed down few more overriding examples.

```
MacBook obj = new MacBook();

obj.myMethod();

// This would call the myMethod() of parent class MacBook
```

```
iPad obj = new iPad();

obj.myMethod();

// This would call the myMethod() of child class iPad
```

```
MacBook obj = new iPad();


obj.myMethod();


// This would call the myMethod() of child class iPad
```

In the third example, the method of the child class is to be executed because the method that needs to be executed is determined by the type of object. Since the object belongs to the child class, the child class version of **myMethod()** is called.

Advantages of Dynamic Polymorphism

1. Dynamic Polymorphism allows Java to support overriding of methods which is central for run-time polymorphism.
2. It allows a class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods.
3. It also allows subclasses to add its specific methods subclasses to define the specific implementation of same.

This was all about different types. Now let's see some important other characteristics of Polymorphism.

Other Characteristics of Polymorphism in Java

In addition to these two main types of polymorphism in Java, there are other characteristics in the Java programming language that exhibits polymorphism like:

- Coercion
- Operator Overloading
- Polymorphic Parameters

Let's discuss some of these characteristics.

Coercion

Polymorphic coercion deals with implicit type conversion done by the compiler to prevent type errors. A typical example is seen in an integer and string concatenation.

```
String str="string"=2;
```

Operator Overloading

An operator or method overloading refers to a polymorphic characteristic of same symbol or operator having different meanings (forms) depending on the context. For example, the plus symbol (+) is used for mathematical addition as well as String concatenation. In either case, only context (i.e. argument types) determines the interpretation of the symbol.

```
String str = "2" + 2;
```

```
int sum = 2 + 2;
```

```
System.out.println(" str = %s\n sum = %d\n", str, sum);
```

Output:

```
str = 22
```

```
sum = 4
```

Polymorphic Parameters

Parametric polymorphism allows a name of a parameter or method in a class to be associated with different types. In the below example I have defined *content* as a *String* and later as an *Integer*:

```
public class TextFile extends GenericFile{

private String content;

public String setContentDelimiter(){

int content = 100;

this.content = this.content + content;

}

}
```

Note: *Declaration of polymorphic parameters can lead to a problem known as variable hiding***.**

Here, the local declaration of a parameter always overrides the global declaration of another parameter with the same name. To solve this problem, it is often advisable to use global references such as *this* keyword to point to global variables within a local context.