Artículo I

Herencia en Java con ejemplos

Swatee Chand

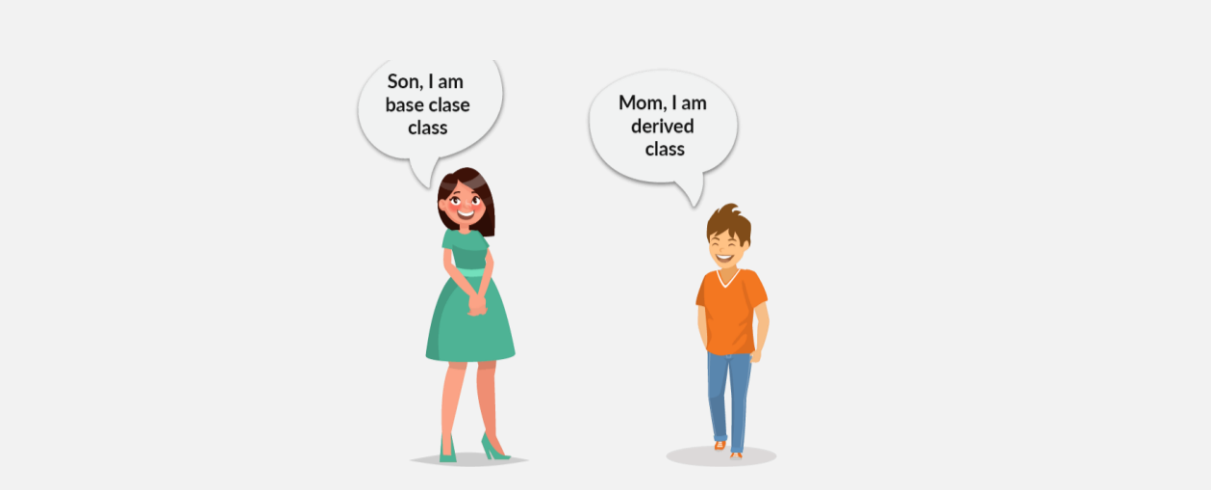
La programación orientada a objetos o mejor conocida como OOPs es uno de los principales pilares de Java que ha aprovechado su potencia y facilidad de uso. Para convertirse en un desarrollador profesional de Java, debe obtener un control impecable sobre los diversos conceptos de OOPs de Java como herencia, abstracción, encapsulación y polimorfismo. A través de este artículo, le daré una visión completa de uno de los conceptos más importantes de OOPs, es decir, la herencia en Java y cómo se logra.

A continuación, se presentan los temas, voy a discutir en este artículo:

- Introducción a la herencia en Java
- Tipos de herencia en Java
  - Herencia única
  - Herencia multinivel
  - Herencia jerárquica
  - Herencia híbrida
  - Reglas de herencia en Java

Introducción a la herencia en Java

En OOP, los programas de computadora están diseñados de tal manera que todo es un objeto que interactúa entre sí. La herencia es una parte integral de los OOPs java que permite que las propiedades de una clase sean heredadas por la otra. Básicamente, ayuda a reutilizar el código y establecer una relación entre diferentes clases.



Como sabemos, un niño hereda las propiedades de sus padres. Un concepto similar se sigue en Java, donde tenemos dos clases:

1. Clase padre (Super o Clase base)

2. Clase secundaria (subclase o clase derivada)\

Una clase que hereda las propiedades se conoce como clase secundaria, mientras que una clase cuyas propiedades se heredan se conoce como clase parent.

Sintaxis:

Ahora, para heredar una clase necesitamos usar la palabra clave *extends.* En el ejemplo siguiente, la clase Son es la clase secundaria y la clase Mom es la clase principal. La clase Son hereda las propiedades y métodos de la clase Mom.

```
class Son extends Mom
{
//your code
}
```

Veamos un pequeño programa y entendamos cómo funciona. En este ejemplo, tenemos una clase base Teacher y una subclase HadoopTeacher. Dado que la clase HadoopTeacher extiende las propiedades de la clase base, no necesitamos declarar estas propiedades y método en la subclase.

```
class Teacher{
  String designation = "Teacher";
  String collegeName = "Edureka";
  void does(){
    System.out.println("Teaching");
  }
}

public class HadoopTeacher extends Teacher{
  String mainSubject = "Spark";
  public static void main(String args[]){
  HadoopTeacher obj = new HadoopTeacher();
  System.out.println(obj.collegeName);
  System.out.println(obj.designation);
  System.out.println(obj.mainSubject);
  obj.does();
  }
}
```

Output:

Edureka

Teacher

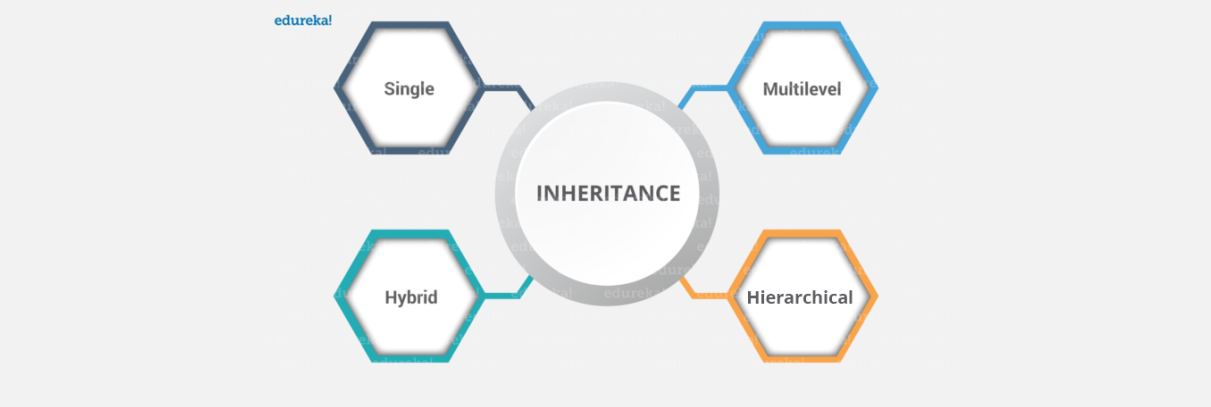
Spark

Teaching

Ahora vamos a ir más allá y ver los diversos tipos de herencia compatibles con Java.

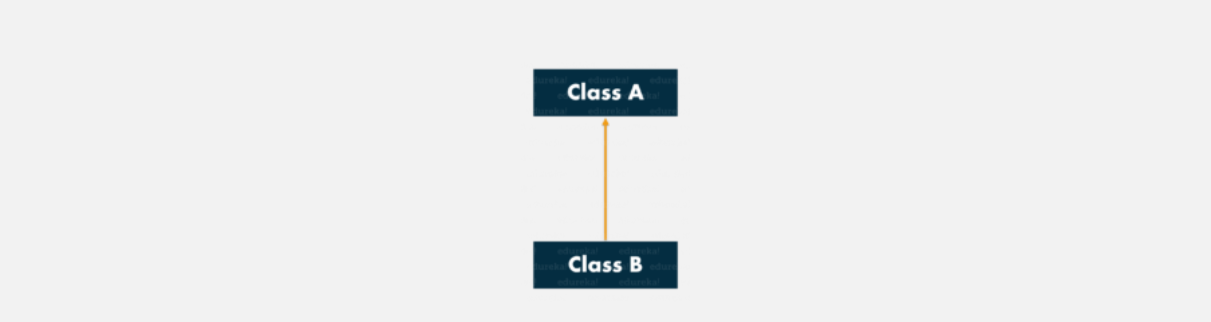
Tipos de herencia en Java

La siguiente figura muestra los tipos de herencia:



Herencia única

En la herencia única, una clase hereda las propiedades de otra. Permite a una clase derivada heredar las propiedades y el comportamiento de una sola clase primaria. Esto, a su vez, habilitará la reutilización del código, así como agregará nuevas características al código existente.



Aquí, la Clase A es su clase primaria y la Clase B es su clase secundaria que hereda las propiedades y el comportamiento de la clase primaria. Un concepto similar se representa en el siguiente código:

```
class Animal{
  void eat(){System.out.println("eating");}
}
class Dog extends Animal{
  void bark(){System.out.println("barking");}
}
class TestInheritance{
  public static void main(String args[]){
  Dog d=new Dog();
  d.bark();
  d.eat();
}
}
```

Herencia multinivel

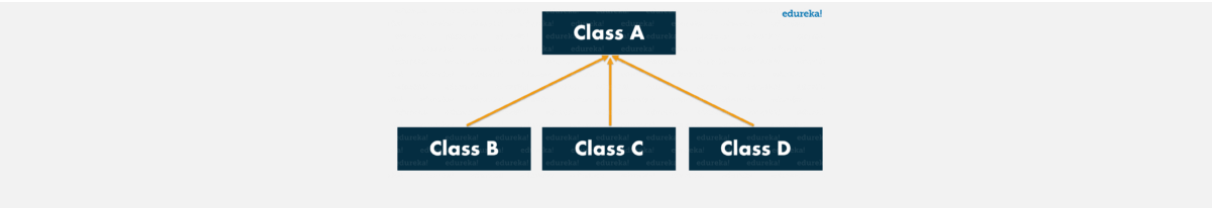
Cuando una clase se deriva de una clase que también se deriva de otra clase, es decir, una clase que tiene más de una clase primaria pero en niveles diferentes, este tipo de herencia se denomina herencia multinivel.

Si hablamos del diagrama de flujo, la clase B hereda las propiedades y el comportamiento de la clase A y la clase C hereda las propiedades de la clase B. Aquí A es la clase primaria para B y la clase B es la clase padre para C. Por lo tanto, en este caso, la clase C hereda implícitamente las propiedades y métodos de la clase A junto con la clase B. Eso es lo que es la herencia multinivel.

```
class Animal{
  void eat(){System.out.println("eating…");}
}
class Dog extends Animal{
  void bark(){System.out.println("barking…");}
}
class Puppy extends Dog{
  void weep(){System.out.println("weeping…");}
}
class TestInheritance2{
  public static void main(String args[]){
  Puppy d=new Puppy();
  d.weep();
  d.bark();
  d.eat();
}
}
```

Herencia jerárquica

Cuando una clase tiene más de una clase secundaria (subclases) o, en otras palabras, más de una clase secundaria tiene la misma clase padre, entonces este tipo de herencia se conoce como **jerárquica**.

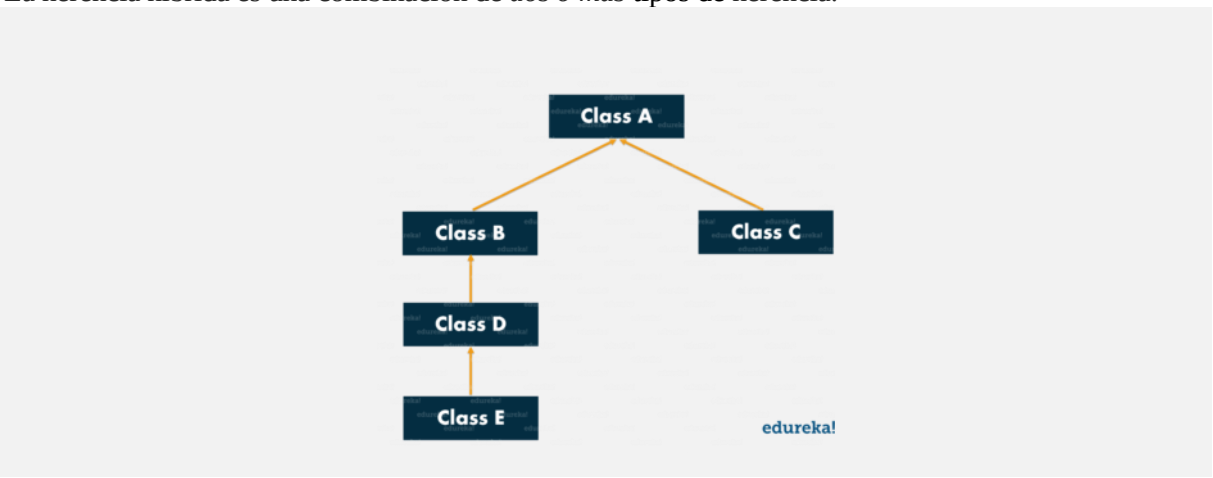


En el diagrama anterior, la Clase B y C son las clases secundarias que heredan de la clase padre, es decir, la Clase A.

```
class Animal{
  void eat(){System.out.println("eating…");}
}
class Dog extends Animal{
  void bark(){System.out.println("barking…");}
}
class Cat extends Animal{
  void meow(){System.out.println("meowing…");}
}
class TestInheritance3{
  public static void main(String args[]){
    Cat c=new Cat();
    c.meow();
    c.eat();
  }
}
```

Herencia híbrida

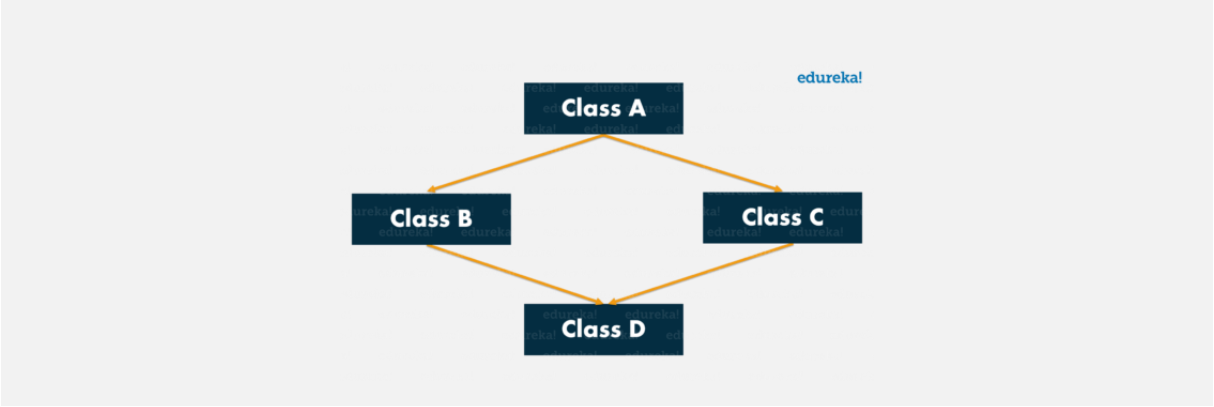
La herencia híbrida es una combinación de *dos o más* tipos de herencia.



Ahora que sabemos lo que es la herencia y sus diversos tipos, vamos a ir más allá y ver algunas de las reglas importantes que se deben tener en cuenta al heredar clases.

Reglas de herencia en Java

REGLA 1: La herencia múltiple NO está permitida en Java.



La herencia múltiple hace referencia al proceso en el que una clase secundaria intenta extender más de una clase primaria. En la ilustración anterior, la clase A es una clase primaria para la clase B y C, que se extienden aún más por la clase D. Esto es resultados en el problema del diamante. ¿por qué? Supongamos que tiene un método show() en las clases B y C, pero con diferentes funcionalidades. Cuando la clase D extiende la clase B y C, hereda automáticamente las características de B y C, incluido el método show(). Ahora, cuando intenta invocar show() de la clase B, el compilador se confundirá en cuanto a qué show() se invocará (ya sea desde la clase B o la clase C). Por lo tanto, conduce a la ambigüedad.

por ejemplo:

```
class Demo1
{
//code here
}
class Demo2
{
//code here
}
class Demo3 extends Demo1, Demo2
{
//code here
}
class Launch
{
public static void main(String args[])
{
//code here
```

```
}
}
```

En el código anterior, Demo3 es una clase secundaria que está intentando heredar dos clases primarias Demo1 y Demo2. Esto no está permitido, ya que da lugar a un problema de diamantes y conduce a la ambigüedad.

NOTA: La herencia múltiple no está soportada en Java, pero aún puede lograrla mediante interfaces.

REGLA 2: La herencia cíclica NO está permitida en Java.

Es un tipo de herencia en el que una clase se extiende y forma un bucle. Ahora piense si una clase se extiende por sí misma o de alguna manera, si forma un ciclo dentro de las clases definidas por el usuario, entonces hay alguna posibilidad de extender la clase Object. Esa es la razón por la que no está permitido en Java.

por ejemplo:

```
class Demo1 extends Demo2
{
//code here
}
class Demo2 extends Demo1
{
//code here
}
```

En el código anterior, ambas clases están tratando de heredar los caracteres de la otra, lo que no está permitido, ya que conduce a la ambigüedad.

REGLA 3: Los miembros privados NO se heredan.

por ejemplo:

```
class You
{
  private int an;
  private int pw;
  You(){
       an =111;
```

```
      pw= 222;
   }
}

class Friend extends You
{
  void changeData()
  {
    an =8888;
    pw=9999;
  }
  void disp()
  {
    System.out.println(an);
    System.out.println(pw);
  }
}
class Launch
{
  public static void main(String args[])
  {
    Friend f = new Friend();
    f.changeData();
    f.disp();
  }
}
```

Cuando ejecute el código anterior, adivine lo que sucede, ¿cree que las variables privadas *an* y *pw* se heredarán? en absoluto. Sigue siendo el mismo porque son específicos de la clase en particular.

REGLA 4: Los constructores no se pueden heredar en Java.

Un constructor no se puede heredar, ya que las subclases siempre tienen un nombre diferente.

```
class A {
   A();
}

class B extends A{
   B();
}
```

Sólo puede hacer lo siguiente:

```
B b = new B(); // and not new A()
```

Los métodos, en su lugar, se heredan con "el mismo nombre" y se pueden utilizar. Sin embargo, aún puede usar constructores de A dentro de la implementación de B:

```
class B extends A{
  B(){
   super();
  }
}
```

REGLA 5: En Java, asignamos referencia padre a objetos secundarios.

Parent es una referencia a un objeto que resulta ser un subtipo de Parent, es decir, un objeto secundario. *¿Por qué se usa esto?* Bueno, en resumen, evita que el código se acopla estrechamente con una sola clase. Puesto que la referencia es de una clase primaria, puede contener cualquiera de sus objetos de clase secundaria, es decir, puede hacer referencia a cualquiera de sus clases secundarias.

Tiene las siguientes ventajas:-

1. La distribución dinámica de métodos permite a Java admitir la anulación de un método, que es fundamental para el polimorfismo en tiempo de ejecución.

2. Permite que una clase especifique métodos que serán comunes a todos sus derivados, al tiempo que permite que las subclases definan la implementación específica de algunos o todos esos métodos.

3. También permite a las subclases agregar sus subclases de métodos específicos para definir la implementación específica de algunas.

Imagine que agrega getEmployeeDetails a la clase Parent como se muestra en el siguiente código:

```
public String getEmployeeDetails() {
    return "Name: " + name;
}
```

Podríamos invalidar ese método en Child para proporcionar más detalles.

```
@Override
public String getEmployeeDetails() {
```

```
    return "Name: " + name + " Salary: " + salary;
}
```

Ahora puede escribir una línea de código que obtenga los detalles disponibles, ya sea que el objeto sea primario o secundario, como:

```
parent.getEmployeeDetails();
```

A continuación, compruebe el código siguiente:

```
Parent parent = new Parent();
parent.name = 1;
Child child = new Child();
child.name = 2;
child.salary = 2000;
Parent[] employees = new Parent[] { parent, child };
for (Parent employee : employees) {
    employee.getEmployeeDetails();
}
```

Esto dará como resultado el siguiente resultado:

```
Name: 1
Name: 2 Salary: 2000
```

Aquí hemos utilizado una clase Child como referencia de clase Parent. Tenía un comportamiento especializado que es único para la clase Child, pero si invocamos getEmployeeDetails(), podemos ignorar la diferencia de funcionalidad y centrarnos en cómo las clases Parent y Child son similares.

REGLA 6: Los constructores se ejecutan debido a super() presente en el constructor.

Como ya sabe, los constructores no se heredan, pero se ejecuta debido a la palabra clave super(). 'super()' se utiliza para referirse a la clase extendida. De forma predeterminada, hará referencia a la clase Object. El constructor de Object no hace nada. Si un constructor no invoca explícitamente un constructor de superclase, el compilador de Java insertará una llamada al constructor sin argumentos de la superclase de forma predeterminada.

Esto nos lleva al final de este artículo sobre "Herencia en Java". Espero, lo encontró informativo y ayudó a agregar valor a su conocimiento.

Artículo II

Polimorfismo maestro en Java con ejemplos

Swatee Chand

En el mundo real, es posible que haya visto un camaleón cambiando su color según sus requisitos. Si alguien pregunta: "¿Cómo lo hace?", simplemente puede decir: "Porque es polimórfico". Del mismo modo, en el mundo de la programación, los objetos Java poseen la misma funcionalidad donde cada objeto puede tomar múltiples formas. Esta propiedad se conoce como Polimorfismo en Java, donde Poly significa muchos y morph significa cambio (o 'forma'). En este artículo, vamos a discutir este concepto clave de programación orientada a objetos, es decir, polimorfismo en Java.

A continuación se muestran los temas que se tratarán en este artículo:

1. ¿Qué es el polimorfismo?
2. Polimorfismo en Java con ejemplo
3. Tipos de Polimorfismo en Java

    a) Polimorfismo estático

b) Polimorfismo dinámico

c) Other Characteristics of Polymorphism in Java

¿Qué es el polimorfismo?

El polimorfismo es la capacidad de una entidad para tomar varias formas. En la programación orientada a objetos, se refiere a la capacidad de un objeto (o una referencia a un objeto) para tomar diferentes formas de objetos. Permite enviar un mensaje común de recopilación de datos a cada clase. El polimorfismo fomenta la llamada como 'extendibilidad', lo que significa que un objeto o una clase puede tener sus usos extendidos.

En la figura anterior, se puede ver, *el hombre* es sólo uno, pero él toma múltiples roles como - él es un padre de su hijo, él es un empleado, un vendedor y muchos más. Esto se conoce como ***Polimorfismo.***

Ahora, vamos a entender esto tomando un ejemplo de la vida real y ver cómo este concepto encaja en la programación orientada a objetos.

Polimorfismo en Java con ejemplo

Vamos a tratar el siguiente ejemplo:.

Considere la posibilidad de un teléfono celular donde guardar sus contactos. Supongamos que una persona tiene dos números de contacto. Para facilitar la accesibilidad, su teléfono celular le proporciona la funcionalidad donde puede guardar dos números bajo el mismo nombre.

Del mismo modo, en Java, un objeto es sólo uno, pero puede tomar varias formas dependiendo del contexto del programa. Supongamos que desea escribir una función para guardar dos números de contacto de la misma persona, puede crearla como — **void createContact(String name, int number1, int number2)**.

Ahora, no es necesario que todos en su lista de contactos tengan dos números de contacto. Pocos de ellos podrían tener un solo número de contacto. En tales situaciones, en lugar de crear otro método con un nombre diferente para guardar un número para un contacto, lo que puede hacer es, crear otro método con el mismo nombre, es decir, **createContact().** Pero, en lugar de tomar dos números de contacto como parámetros, tome solo un número de contacto como parámetro, es decir, **void createContact(String name, int number1)**.

Como puede ver en la figura anterior, el método **createContact()** tiene dos definiciones diferentes. Aquí, la definición que se va a ejecutar depende del número de parámetros que se pasan. Si se pasa un parámetro, solo se guarda un número de contacto en el contacto. Pero, si se pasan dos números de

contacto a este método al mismo tiempo, ambos se guardarán en el mismo contacto. *Esto también se conoce como* sobrecarga de ***métodos.***

Ahora tomemos otro ejemplo y entendamos el polimorfismo en profundidad.

Supongamos que fue a un centro comercial (Allen Solly) cerca de su casa y compró un par de pantalones vaqueros. Una semana más tarde, mientras viaja a una ciudad cercana, verá otro centro comercial. Entras en la tienda y encuentras una nueva variante de la misma marca que te gustó aún más. Pero decidiste comprarlo en la tienda cercana a tu casa. Una vez de vuelta a casa, nuevamente fue al centro comercial cerca de su casa para obtener esos increíbles pares de jeans, pero no pudo encontrarlo. ¿por qué? Porque esa era una especialidad de la tienda que se ubicaba en el pueblo vecino.

Ahora, relacionando este concepto con un lenguaje orientado a objetos como Java, supongamos que tiene una clase llamada XJeans que incluye un método llamado **jeans().** Usando este método, puede obtener unos jeans Allen Solly. Para los Jeans en la ciudad vecina, hay otra clase **YJeans.** Tanto las clases XJeans como YJeans amplían la clase padre **ABCShoppingCenter.** La clase YJeans incluye un método denominado **jeans()**, con el que puede obtener ambas variantes de jeans.

```
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");
  }
}
```

Por lo tanto, en lugar de crear diferentes métodos para cada nueva variante, podemos tener un solo método de jeans**()**, que se puede definir según las diferentes clases secundarias. Por lo tanto, el método llamado **jeans()** tiene dos definiciones: una con solo jeans predeterminados y otra con ambas, los jeans predeterminados y la nueva variante. Ahora, el método que se invoca dependerá del tipo de objeto al que pertenece. Si crea el objeto de clase **ABCShoppingCenter,** solo habrá un jeans disponible. Pero si crea el objeto de clase **YJeans,** que extiende la clase **ABCShoppingCenter,** puede tener ambas variantes. *Esto también se conoce como* reemplazo de ***métodos.*** Por lo tanto, el polimorfismo aumenta la simplicidad y legibilidad del código al reducir la complejidad. Esto hace que el polimorfismo en Java sea un concepto muy útil y también se puede aplicar en escenarios del mundo real.

Espero que os hayas hecho una idea sobre el concepto de Polimorfismo. Ahora, vamos a ir más allá con este artículo y entender diferentes tipos de **polimorfismo en Java**.

Tipos de Polimorfismo en Java

Java soporta dos tipos de polimorfismo y son los siguientes:

- Polimorfismo estático
- Polimorfismo dinámico

Polimorfismo estático

Un polimorfismo que se resuelve durante el tiempo de compilación se conoce como polimorfismo estático. La sobrecarga de métodos es un ejemplo de polimorfismo en tiempo de compilación.

Ejemplo

La sobrecarga de métodos es una característica que permite que una clase tenga dos o más **métodos** que tengan el mismo nombre, pero con listas de parámetros diferentes. En el ejemplo siguiente, tiene dos definiciones del mismo método add(). Por lo tanto, el método add() al que se llamaría viene

determinado por la lista de parámetros en tiempo de compilación. Esa es la razón por la que esto también se conoce como polimorfismo en tiempo de compilación.

```
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));
  }
}
```

Así es como funciona el Polimorfismo Estático. Ahora, vamos a entender lo que es el polimorfismo dinámico en Java.

Polimorfismo dinámico

El polimorfismo dinámico es un proceso en el que una llamada a un método reemplazado se resuelve en tiempo de ejecución, por eso se denomina polimorfismo en tiempo de ejecución. La invalidación de métodos es una de las formas de lograr el polimorfismo dinámico. En cualquier lenguaje de programación orientado a objetos, **Overriding** es una característica que permite a una subclase o clase secundaria proporcionar una implementación específica de un **método** que ya ha sido proporcionado por una de sus superclases o clases primarias.

Ejemplo

En el siguiente ejemplo, tienes dos clases **De MacBook** y **iPad.** *MacBook* es una clase principal y el *iPad* es una clase secundaria. La clase secundaria está reemplazando el método **myMethod()** de la

clase padre. Aquí, he asignado el objeto de clase secundaria a la referencia de clase primaria para determinar qué método se llamaría en tiempo de ejecución. Este es el tipo de objeto que determina a qué versión del método se llamaría (no el tipo de referencia).

```
class MacBook{
  public void myMethod(){
    System.out.println("Overridden Method");
  }
}

public class iPad extends MacBook{
  public void myMethod(){
    System.out.println("Overriding Method2");
  }
  public static void main(String args[]){
    MacBook obj = new iPad();
    obj.myMethod();
  }
}
```

Salida: Overriding Method

Cuando se invoca el método de reemplazo, el objeto determina qué método se va a ejecutar. Por lo tanto, esta decisión se toma en tiempo de ejecución.

He enumerado algunos ejemplos más de overriding (sobreescritura).

```
MacBook obj = new MacBook();
obj.myMethod();
// Esto llamaría a myMethod() de la clase padre MacBook

iPad obj = new iPad();
obj.myMethod();
// Esto llamaría a myMethod() de la clase secundaria iPad

MacBook obj = new iPad();
obj.myMethod();
// Esto llamaría a myMethod() de la clase secundaria iPad
```

Salida:
Overriding Method
Overriding Method2
Overriding Method2

En el tercer ejemplo, el método de la clase secundaria se va a ejecutar porque el método que se debe ejecutar viene determinado por el tipo de objeto. Puesto que el objeto pertenece a la clase secundaria, se llama a la versión de la clase secundaria de **myMethod().**

Ventajas del Polimorfismo Dinámico

1. El polimorfismo dinámico permite a Java admitir la anulación de métodos, que es fundamental para el polimorfismo en tiempo de ejecución.

2. Permite que una clase especifique métodos que serán comunes a todos sus derivados, al tiempo que permite que las subclases definan la implementación específica de algunos o todos esos métodos.

3. También permite a las subclases agregar sus subclases de métodos específicos para definir la implementación específica de las mismas.

Se tratado de diferentes tipos. Ahora veamos algunas otras características importantes del Polimorfismo.

Otras características del polimorfismo en Java

Además de estos dos tipos principales de polimorfismo en Java, hay otras características en el lenguaje de programación Java que exhibe polimorfismo como:

1. Coacción (Coercion)
2. Sobrecarga de operadores
3. Parámetros polimórficos

Vamos a discutir algunas de estas características.

Coacción

La conversión polimórfica se ocupa de la conversión implícita de tipos realizada por el compilador para evitar errores de tipo. Un ejemplo típico se ve en una concatenación de enteros y cadenas.

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

Sobrecarga (Overloading) de operadores

Una sobrecarga de operador o método se refiere a una característica polimórfica del mismo símbolo u operador que tiene diferentes significados (formas) dependiendo del contexto. Por ejemplo, el símbolo más (+) se utiliza para la suma matemática, así como la concatenación de cadenas. En cualquier caso, solo el contexto (es decir, los tipos de argumento) determina la interpretación del símbolo.

```
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

El polimorfismo paramétrico permite asociar un nombre de un parámetro o método de una clase a diferentes tipos. En el siguiente ejemplo he definido *el contenido* como una *cadena* y más tarde como un *entero:*

```
public class TextFile extends GenericFile{
  private String content;
  public String setContentDelimiter(){
    int content = 100;
    this.content = this.content + content;
  }
}
```

Nota: *La declaración de parámetros polimórficos puede conducir a un problema conocido como ocultación de variables.*

Aquí, la declaración local de un parámetro siempre invalida la declaración global de otro parámetro con el mismo nombre. Para resolver este problema, a menudo es aconsejable utilizar referencias globales como *this* palabra clave para apuntar a variables globales dentro de un contexto local.

Con esto, llegamos a su fin este artículo sobre el artículo Polimorfismo en Java. Busque otros artículos de la serie que le ayudarán a entender varios conceptos de Java.

Artículo III

Polymorphism/Abstract Methods

Problem

Desea que cada una de varias subclases proporcione su propia versión de uno o más métodos.

Solution

Haga que el método sea abstracto en la clase primaria; esto hace que el compilador se asegure de que cada subclase lo implementa.

Discussion

Un programa de dibujo hipotético utiliza una subclase Shape para cualquier cosa que se dibujó. Shape tiene un método abstracto denominado computeArea() que calcula el área exacta de la forma dada:

```
public abstract class Shape {
    protected int x, y;
    public abstract double computeArea( );
}
```

Una subclase Rectangle, por ejemplo, tiene un computeArea() que multiplica el ancho por el alto y devuelve el resultado:

```
public class Rectangle extends Shape {
    double width, height;
    public double computeArea( ) {
        return width * height;
    }
}
```

Una subclase Circle devuelve $\pi r^{2:}$

```
public class Circle extends Shape {
    double radius;
    public double computeArea( ) {
        return Math.PI * radius * radius;
    }
}
```

Este sistema tiene un alto grado de generalidad. En el programa principal, podemos iterar sobre una colección de objetos Shape y, aquí está la verdadera belleza, llamar a computeArea() en cualquier objeto de subclase Shape sin tener que preocuparnos por qué tipo de forma es. Los métodos polimórficos de Java llaman automáticamente al método computeArea() correcto en la clase de la que se construyó originalmente el objeto:

main/src/main/java/oo//shapes/ShapeDriver.java

```
/** Parte de un programa principal usando objetos Shape */
public class ShapeDriver {
```

```
Collection<Shape> allShapes;    // creado en un constructor, no se muestra

public double totalAreas() {
    double total = 0.0;
    for (Shape s : allShapes) {
        total += s.computeArea();
    }
    return total;
}
```

El polimorfismo es una gran bendición para el mantenimiento del software: si se agrega una nueva subclase, el código en el programa principal no cambia. Además, todo el código que es específico de, por ejemplo, el control de polígonos, está todo en un solo lugar: en el archivo de código fuente de la clase Polygon. Esta es una gran mejora con respecto a los lenguajes más antiguos, donde los campos de tipo en una estructura se usaban con instrucciones case o switch dispersas por todo el software. Java hace que el software sea más confiable y mantenible con el uso de polimorfismo.

Fuente: Java CookBook 4ta. Ed. *by O'Reilly Media, Inc., 2020*