Universidad César Vallejo

Pregrado

SESIÓN 09:

GRASP parte 3: - Polimorfismo. - Indirección. - Notación grafica con UML

Agenda

- Sobrecarga de métodos.
- Constructores
- Constructores por defecto y con parámetros
- Polimorfismo

Sobrecarga de Métodos

- Permite definir dos ó más **métodos que comparten el mismo nombre**.

- Cuando se invoca a un método sobrecargado, Java utiliza el tipo y/o número de parámetros como guía para determinar a que versión del método sobrecargado debe llamar. Por lo tanto, los métodos sobrecargados deben ser diferentes en el tipo y/o número de parámetros.

```java
public class Conversion {
  public String mConver(int n) {
    return (Integer.toString(n));
  }

  public String mConver(long n) {
    return (Long.toString(n));
  }

  public String mConver(float n) {
    return (Float.toString(n));
  }

  public String mConver(double n) {
    return (Double.toString(n));
  }
}
```

Se implementan varios métodos con el mismo nombre, se diferencia por los parámetros y/o tipos de datos utilizados

```java
public class AppSobrecarga {
  public static void main(String[] args) {
    Conversion obj = new Conversion();
    int n1 = 500;
    long n2 = 100000;
    float n3 = 456.459834f;
    double n4 = 8934.42573485720;

    System.out.println("n1 = " + obj.mConver(n1));
    System.out.println("n2 = " + obj.mConver(n2));
    System.out.println("n3 = " + obj.mConver(n3));
    System.out.println("n4 = " + obj.mConver(n4));
  }
}
```

Constructor

- Un *Constructor* es un **método especial en Java** empleado para **inicializar valores en Instancias de Objetos**. Sintácticamente es similar a un método.

- A través de este tipo de métodos es posible generar diversos tipos de *instancias* para la Clase en cuestión; la principal característica de este tipo de métodos es que llevan **el mismo nombre de la clase**

- A continuación se describen varios ejemplos utilizando constructores:

Constructor por defecto

```
public class Empleado {
  public String nombre;
  public double sueldo;
  public int edad;

  // Constructor:
  public Empleado() {
    nombre = "Claudia";
    sueldo = 5000.00;
    edad = 22;
  }
}
```

Se implementa un Constructor con valores de inicio para las instancias de la clase

Cuando se crea e instancia un objeto de una clase se inicializa los valores del constructor

```
public class AppEmpleado {
  public static void main(String[] args) {
    Empleado obj = new Empleado();
    System.out.println("Nombre = " + obj.nombre);
    System.out.println("Sueldo = " + obj.sueldo);
    System.out.println("Edad   = " + obj.edad);
  }
}
```

Constructor con Parámetros

```
public class Empleadol {
  public String nombre;
  public double sueldo;
  public int edad;

  // Constructor:
  public Empleadol(String nombre, double sueldo, int edad) {
    this.nombre = nombre;
    this.sueldo = sueldo;
    this.edad = edad;
  }
}
```

Al instanciar el objeto se **puede enviar valores para inicializar constructor**

```
public class AppEmpleadol {
  public static void main(String[] args) {
    Empleadol obj = new Empleadol("Luis Campos", 1500.50, 35);
    System.out.println("Nombre = " + obj.nombre);
    System.out.println("Sueldo = " + obj.sueldo);
    System.out.println("Edad   = " + obj.edad);
  }
}
```

Sobrecarga de Constructores

```java
public class Arboles {
  public Arboles() {
    System.out.println("Un árbol genérico");
  }

  public Arboles(String tipo) {
    System.out.println("Un árbol tipo " + tipo);
  }

  public Arboles(int altura) {
    System.out.println("Un árbol de " + altura + " metros");
  }

  public Arboles(int altura, String tipo) {
    System.out.println("Un " + tipo + " de " + altura + " metros");
  }
}
```

Se implementa varios constructores con el mismo nombre, se diferencia por los parámetros y/o tipos de datos utilizados

```java
public class AppArboles {
  public static void main(String args[]) {
    Arboles arbol1 = new Arboles(4);
    Arboles arbol2 = new Arboles("Roble");
    Arboles arbol3 = new Arboles();
    Arboles arbol4 = new Arboles(5, "Pino");
  }
}
```

Ejemplo de Sobrecarga de Constructores

```
public class Empleado2 {
  private String nombre;
  private double sueldo;
  private int edad;
  private static final double SUELDO_BASE = 5000.00;
// Constructores:
  public Empleado2(String nombre, double sueldo, int edad) {
    this.nombre = nombre;
    this.sueldo = sueldo;
    this.edad = edad;
  }
  public Empleado2(String nombre, double sueldo) {
    this(nombre, sueldo, 0);
  }
  public Empleado2(String nombre, int edad) {
    this(nombre, SUELDO_BASE, edad);
  }

  public Empleado2(String nombre) {
    this(nombre, SUELDO_BASE);
  }
// Métodos:
  public String getNombre() {
    return nombre;
  }
  public double getSueldo() {
    return sueldo;
  }
  public int getEdad() {
    return edad;
  }
}
```

Atributos

Implementación de varios constructores con el mismo nombre: **SOBRECARGA DE CONSTRUCTORES**

This permite invocar alguna versión de constructor que coincida con los parámetros, ejecutándose automáticamente hasta llegar al principal

```
public class AppEmpleados2 {

  public static void main(String[] args) {
    Empleado2 emp1 = new Empleado2("Gustavo", 15000.0, 30);
    Empleado2 emp2 = new Empleado2("Ricardo", 27);
    Empleado2 emp3 = new Empleado2("Sergio");

    System.out.println("Empleado 1");
    System.out.println("Nombre = " + emp1.getNombre());
    System.out.println("Sueldo = " + emp1.getSueldo());
    System.out.println("Edad   = " + emp1.getEdad());
    System.out.println();

    System.out.println("Empleado 2");
    System.out.println("Nombre = " + emp2.getNombre());
    System.out.println("Sueldo = " + emp2.getSueldo());
    System.out.println("Edad   = " + emp2.getEdad());
    System.out.println();

    System.out.println("Empleado 3");
    System.out.println("Nombre = " + emp3.getNombre());
    System.out.println("Sueldo = " + emp3.getSueldo());
    System.out.println("Edad   = " + emp3.getEdad());
    System.out.println();
  }
}
```

Ejemplos

```java
public class Cuenta {

    private String codigo; //Numero de la cuenta
    private double saldo; //Saldo de la cuenta

    /** Constructor Vacio */
    public Cuenta(){
        this.codigo = null;
        this.saldo = 0;
    }

    /** Constructor sobrecargado */
    public Cuenta(String codigo, double saldo){
        this.codigo = codigo;
        this.saldo = saldo;
    }
```

Cuantas posibles formas podrían implementarse para realizar transferencia de dinero de una cuenta a otra?

```
/** permite hacer una transferencia de una cuenta a otra */
public void transferencia(Cuenta cuenta, double cantidad){
    if(this.saldo>=cantidad){
        this.saldo-=cantidad;
        cuenta.saldo+=cantidad;
    }
}
```

```
/** permite hacer una transferencia de una cuenta a otra */
public void transferencia(Cuenta cuenta){
    if(this.saldo>0){
        cuenta.saldo+=this.saldo;
        this.saldo=0;
    }
}
```

Transfiriendo todo el saldo de la cuenta actual a la cuenta que recibe como parámetro

```java
/**permite hacer la transferencia entre 2 cuentas */
public static void transferencia(Cuenta cuenta1,
        Cuenta cuenta2, double cantidad){
    if(cuenta1.saldo>=cantidad){
        //Se retira de la cuenta1
        cuenta1.retiro(cantidad);
        //Se deposita en la cuenta2
        cuenta2.deposito(cantidad);
    }
```

Transfiriendo una determinada cantidad desde la cuenta1 hasta la cuenta2

POLIMORFISMO

- El concepto de Polimorfismo es uno de los fundamentos para cualquier lenguaje orientado a Objetos, las mismas raíces de la palabra pueden ser una fuerte pista de su significado: ***Poli* = Multiple**, ***morfismo*= Formas** , esto implica que un mismo Objeto puede tomar diversas formas.

- A través del concepto de Herencias ("Inheritance") es posible ilustrar este comportamiento:

Polimorfismo

- Hay varias formas de polimorfismo:
  - Cuando invocamos el mismo nombre de método sobre instancias de distinta clase
  - cuando creamos múltiples constructores
  - cuando vía subtipo asignamos una instancia de una subclase a una referencia a la clase base.
- Cuando creamos una clase derivada, gracias a la relación es-un podemos utilizar instancias de la clase derivada donde se esperaba una instancia de la clase base. También se conoce como principio de subtitución.

Polimorfismo: Ejemplo

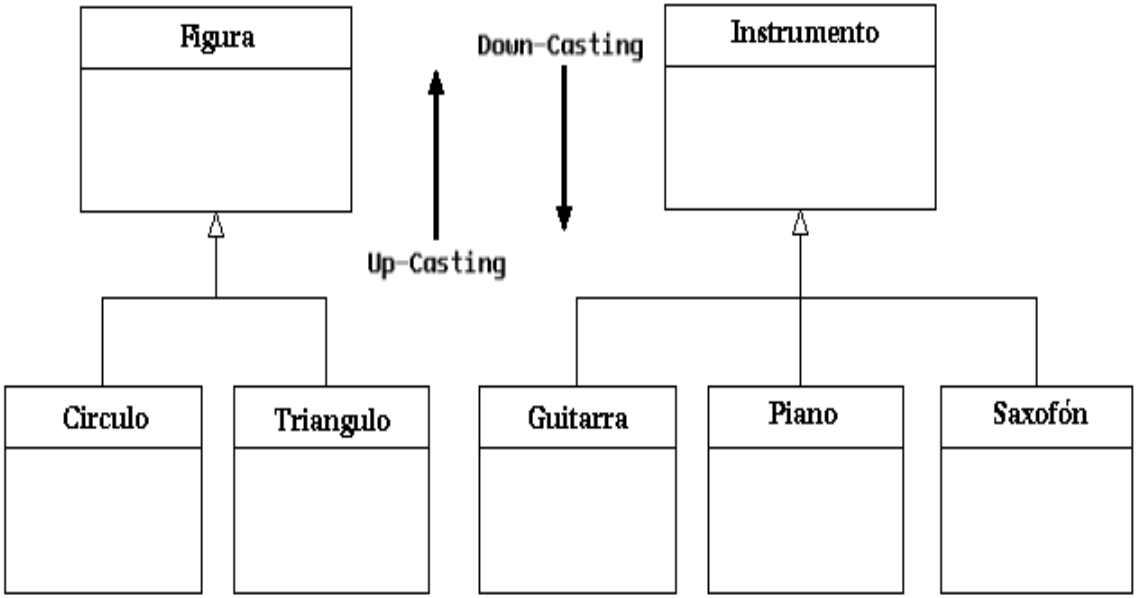
- Sea:

```
class Employee { ..... }
class Manager extends Employee { .... }
Employee e; //declaración de un objeto de Employee
e=new Employee(...); // instancia
e=new Manager(..);   // OK. Subtitución
```

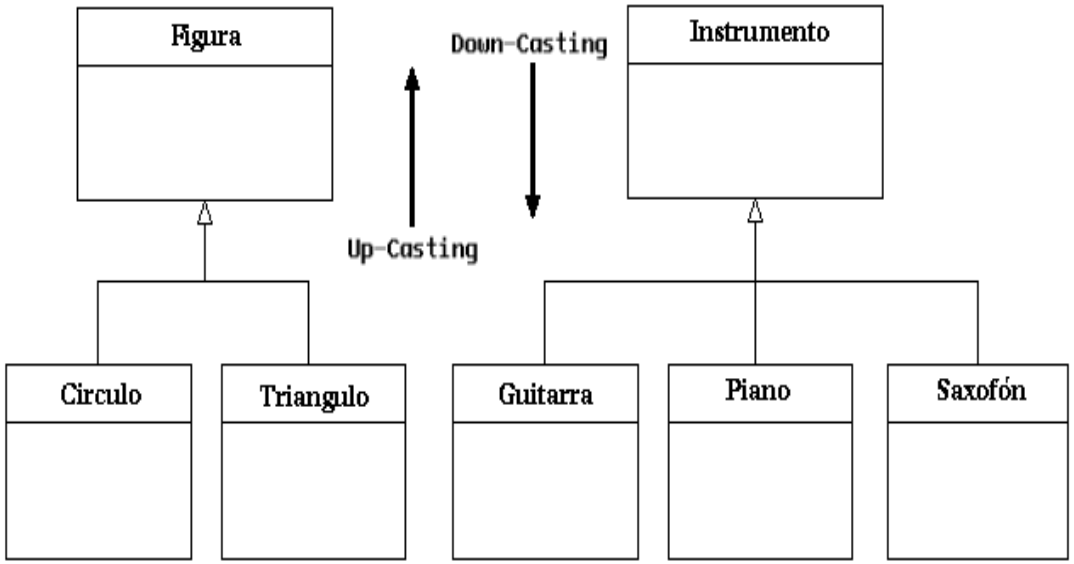
- En el primer caso a través de e tenemos acceso a todo lo correspondiente a un Employee.
- En el segundo caso tenemos acceso a todo lo correspondiente a Employee, pero con la implementación de Manager.
- Al revés no es válido porque toda referencia a Manager debe disponer de todos los campos.

Uso de "Casting”

- El termino "Casting" viene de la palabra "Cast" que significa *Molde*, por lo que el termino literal es *Hacer un Molde*, en Polimorfismo se lleva acabo este proceso de "Casting" implícitamente, una *Guitarra* se coloca en el *molde* de un *Instrumento*, un *Triangulo* en el *molde* de una *Figura*.



Anteriormente se mencionó que el "Casting" llevado acabo con Polimorfismo es implícito, esto se debe a que no se requiere de sintaxis especial, simplemente se convierte una *Guitarra* a un *Instrumento*, sin embargo, para llevar una transformación en sentido opuesto se requiere de sintaxis adicional para mantener la seguridad de transformación; analicemos: mientras se puede asegurar que un *Triangulo* es una *Figura* ("Up-Casting"), pero una *Figura* **no necesariamente** es un *Triangulo*, claro esta que lo puede ser, pero en Java se requiere definir explícitamente esta operación ("Down-Casting").



Polimorfismo con Late Binding

- El poder manipular un Objeto como si éste fuera de un tipo genérico otorga mayor flexibilidad al momento de programar con Objetos, el término Polimorfismo también es asociado con un concepto llamado *Late-Binding* (Ligamiento Tardío), observe el siguiente fragmento de código:

```
Figura a = new Circulo();
Figura b = new Triangulo();
```

Inicialmente se puede pensar que este código generaría un error debido a que el tipo de referencia es distinta a la instancia del objeto, sin embargo, el fragmento anterior es correcto y demuestra el concepto de Polimorfismo; para asentar este tema se describe un ejemplo más completo:

```java
public class Animal {
    public String hacerRuido(){
        return "No definido";
    }

}
```

```java
public class Mamifero extends Animal{
    public String mensaje(){
        return "Soy Mamifero";
    }
}
```

```java
public class Perro extends Mamifero{
    public String mensaje(){
        return "Soy Perro";
    }
    public String hacerRuido(){
        return "guau guau";
    }
}
```

Up-Casting

```java
Mamifero m=new Perro(); //Upcasting
jtxaDatos.append(m.mensaje()+"\n");
jtxaDatos.append(m.hacerRuido()+"\n");
```

Down-Casting

```java
//generando un objeto
 Animal a=new Perro();
//convirtiendo el objeto
Perro nuevoP=(Perro)a; //DOWN-CASTING

jtxaDatos.append(nuevoP.mensaje()+"\n");
jtxaDatos.append(nuevoP.hacerRuido()+"\n");
```

TECNICAS DE CASTING

- Consiste en realizar las conversiones de tipo, no modifican al objeto, solo su tipo.
- **UPCASTING:** permite interpretar un objeto de una clase derivada como del mismo tipo de la clase base. No hace falta especificarlo.
- **DOWNCASTING:** Permite interpretar un objeto de una clase base como del mismo tipo que su clase derivada. Se especifica precediendo al objeto a convertir con el nuevo tipo entre paréntesis.

Operador instanceof

El operador **instanceof** sirve para consultar si un objeto es una instancia de una clase determinada, o de su padre. Se utiliza para evitar hacer casting de objetos a la hora de tratar un objeto de una forma y otra, llamando a un método de una clase o de otra dependiendo de qué tipo de objeto sea. Ejemplo

```
public class Empleado{..}
public class Jefe extends Empleadp{..}
public class Contractor extends Empleado{..}
```

```
public void metodo(Empleado e)
{    if ( e instanceof  Jefe)
          //Obtener beneficios por su salario
     else if (e instanceof  Contractor)
          //obtener tarifa por hora
          else //empleados temporales
}
```

Lo mismo ocurre si en lugar de tener clases y subclases, implementamos interfaces.
Por ejemplo, veamos el siguiente caso:

```
interface Nada {..}
class A implements Nada {..}
class B extends A {..}
….
 A a = new A();
 B b = new B();
....
```

Las siguientes sentencias serían verdaderas

```
a instanceof Nada
b instanceof A
b instanceof Nada //Ya que b implementa el interfaz Nada, de manera indirecta
```

Universidad César Vallejo

Licenciada por Sunedu
para que puedas salir adelante