POO en Java

Sitio:	Centros - Cádiz	Imprimido por:	Barroso López, Carlos
Curso:	Programación	Día:	martes, 21 de mayo de 2024, 23:38
Libro:	POO en Java		

Tabla de contenidos

1. Clases y objetos
1.1. Ejemplo: mejora setContenido()

2. Ámbito

3. Sobrecarga (Overload)

4. Herencia
4.1. Clase abstracta (abstract)
4.2. Sobrescritura (@Override)

5. Interfaces

6. Modificador "static"
6.1. Ejemplo

7. Variables y referencias

8. Casting de objetos

9. Polimorfismo

10. Ejemplo: clase Cubo

1. Clases y objetos

1. Definición de una clase

Sintaxis:

```
public class NombreClase {
  // atributos
  modificador tipo atributo1;
  modificador tipo atributo2;
  ...
  // métodos
  modificador tipo método1(parámetros) { ... }
  modificador tipo método2(parámetros) { ... }
  ...
}
```

Ejemplo:

```
public class Cubo {
  // atributos
  private int capacidad; // capacidad máxima en litros
  private int contenido; // contenido actual en litros

  // métodos
  /**
  * Vacía el cubo.
  */
  public void vacia() {
    this.contenido = 0;
  }
  /**
  * Llena el cubo al máximo de su capacidad.
  */
  public void llena() {
    this.contenido = this.capacidad;
  }
}
```

2. Instanciación

Crea uno o varios objetos de una clase. En Java se utiliza el **operador "new"**.

Ejemplo:

```
Cubo unCubo = new Cubo();
```

3. Métodos especiales

3.1. Constructor

Este método es llamado cuando se crea un objeto (instanciación), y habitualmente se utiliza para **inicializar sus atributos**. Si no es definido, el compilador crea uno por defecto.

El método constructor debe nombrarse igual que la clase y no tiene tipo.

Ejemplo:

```
// Método constructor
Cubo (int c) {
  this.capacidad = c;
}
```

```
// Instanciación
Cubo cubo1 = new Cubo(4);
```

Nota: El **operador "this"** hace referencia al propio objeto, y se utiliza habitualmente para referenciar a sus miembros (atributos y métodos).

3.2. Getter y Setter

Se utilizan para obtener (*get*) o establecer (*set*) el valor de un cierto atributo, y de esta forma, no acceder a dicho atributo de manera directa (encapsulación).

Ejemplo:

Definición (clase)

```
// métodos getter
int getCapacidad() {
  return this.capacidad;
}
int getContenido() {
  return this.contenido;
}
```

```
// método setter
void setContenido(int litros) {
  this.contenido = litros;
}
```

Utilización (objeto)

```
cubo1.setContenido(3);
System.out.println( "Capacidad: " + cubo1.getCapacidad() );
System.out.println( "Contenido: " + cubo1.getContenido() );
```

1.1. Ejemplo: mejora setContenido()

Fichero: Cubo.java

```
// Método setContenido()
int setContenido(int litros) {
    int respuesta = 0;
    if (litros > this.capacidad){
        this.contenido = this.capacidad;
        respuesta = 1;
    }
    else if(litros < 0){
        this.contenido = 0;
        respuesta = -1;
    }
    else{
        this.contenido = litros;
    }
    return respuesta;
}
```

Fichero: PruebaCubo.java

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

        Cubo cubo1, cubo2;
        cubo1 = new Cubo(5);
        cubo2 = new Cubo(10);
        cubo1.setContenido(3);
        cubo2.setContenido(8);
        System.out.println("Capacidad cubo1 " + cubo1.getCapacidad());
        System.out.println("Contenido cubo1 " + cubo1.getContenido());
        System.out.println("Capacidad cubo2 " + cubo2.getCapacidad());
        System.out.println("Contenido cubo2 " + cubo2.getContenido());
        // Chequeo valor devuelto por setContenido()
        System.out.println(cubo1.setContenido(6));  // Salida: 1
        System.out.println(cubo1.setContenido(2));  // Salida: 0
        System.out.println(cubo1.setContenido(-4));  // Salida: -1
    }
}
```

2. Ámbito

1. Modificadores public, protected y private

Al definir los miembros de una clase, se pueden especificar sus ámbitos de visibilidad o accesibilidad con las palabras reservadas: **public** (público), **protected** (protegido) y **private** (privado)

En Java actúan según la siguiente tabla:

	En la misma clase	En el mismo paquete	En una subclase	Fuera del paquete
private	✔	✖	✖	✖
protected	✔	✔	✔	✖
public	✔	✔	✔	✔
sin especificar	✔	✔	✖	✖

Como **regla general**, se suelen declarar private los atributos o variables de instancia y public los métodos.

2. Ejemplos

Caso 1:

```
public class Cubo {
  private int capacidad;
  ...
}
```

```
System.out.println( "Capacidad: " + cubo1.capacidad );  // error
```

Caso 2:

```
public class Cubo {
  public int capacidad;
  ...
}
```

```
System.out.println( "Capacidad: " + cubo1.capacidad );  // ok
```

Pregunta: ¿Qué ocurre cuando se usa el modificador protected? ¿Y si no se utiliza ningún modificador?

3. Sobrecarga (Overload)

1. Sobrecarga de métodos

Al igual que las funciones, los métodos de una clase pueden estar sobrecargados, es decir, un mismo método puede tener más de una implementación utilizando **diferentes parámetros**.

La sobrecarga del método constructor es una práctica muy habitual.

Ejemplo:

```
public abstract class Animal {  // clase abstracta Animal
  private char sexo;
  public Animal () {  // constructor 1: sin parámetros
    sexo = 'M';
  }
  public Animal (char s) {  // constructor 2: 1 parámetro (char)
    sexo = s;
  }
}
```

Tipos de sobrecarga:

1.1. Diferente número de parámetros

Ejemplo:

```
public class Multiplicador {

    public int multiplicar(int a, int b) {
        return a * b;
    }

    public int multiplicar(int a, int b, int c) {
        return a * b * c;
    }
}
```

1.2. Diferente tipo de parámetros

Ejemplo:

```
public class Multiplicador {

    public int multiplicar(int a, int b) {
        return a * b;
    }

    public double multiplicar(double a, double b) {
        return a * b;
    }
}
```

Es posible combinar ambos tipos de sobrecargas.

Sin embargo, **no es posible** tener diferentes implementaciones que difieran solo en el tipo de retorno.

Ejemplo:

```
public int multiplicar(int a, int b) {
    return a * b;
}

public double multiplicar(int a, int b) {
    return a * b;
}
```

2. Vinculación estática

La capacidad de asociar una llamada de método específica al cuerpo del método se conoce como *vinculación*.

En el caso de la sobrecarga de métodos, la vinculación se realiza estáticamente **en tiempo de compilación**, comprobando las firmas de los métodos (configuración de parámetros que utilizan).

4. Herencia

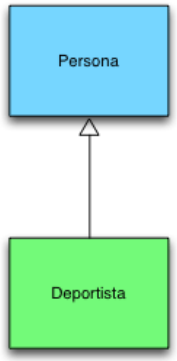
1. Definición de la clase

El mecanismo de herencia permite definir una clase que tendrá disponibles los atributos y los métodos de la clase de la que hereda (*superclase*).

Sintaxis:

```
public class NombreClase extends SuperClase { ... }
```

Ejemplo:




```
public class Deportista extends Persona { ... }
```

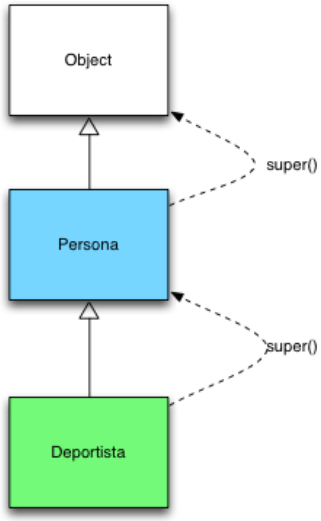
2. Constructor

Como cualquier clase, si no se define ningún método constructor se crea uno por defecto, el cual llama al constructor de la clase padre. Esto es debido a que los constructores no se heredan entre jerarquías de clases.

```
public Deportista() {  // Código implícito, si no se define ningún constructor
  super();  // Llamada al constructor de la clase padre
}
```

2.1. Método super()

Invoca al constructor de la clase base que comparta el mismo tipo de parametrización.



Ejemplo:

Llamada al constructor de la superclase con parámetros.

```
public class CuboPequenio extends Cubo {
    // Capacidad 5L
    public CuboPequenio() {
        super(5);
    }
}
```

3. Otros usos de "super"

3.1. Llamada a los métodos de la superclase redefinidos en la subclase

El método super() dentro de un método de la clase permite llamar al método equivalente de la superclase.

También es posible llamarlo desde cualquier otro método mediante:

```
super.metodo()
```

Ejemplo:

```
public class Sound {
  public void voice() {
    System.out.println("Play sound!");
  }
}
class Drum extends Sound {
  public void voice() {
    System.out.println("Play drums!");
  }
  public void play() {
    super.voice();  // Método del padre
    voice();
  }
}
```

3.2. Acceso a atributos de la superclase cuando la clase tiene atributos con el mismo nombre

```
super.atributo
```

Ejemplo:

```
class Vehicle {
  String name = "vehicle";
}
class Car extends Vehicle {
  String name = "car";
  public void printMyName() {
    System.out.println(name);
  }
  public void printParentName() {
    System.out.println(super.name);
  }
}
```

4.1. Clase abstracta (abstract)

1. Definición

Una clase abstracta es aquella que no va a tener instancias (objetos) de forma directa, aunque sí habrá instancias de las subclases.

Una clase abstracta debe poseer al menos un **método abstracto**.

Sintaxis:

```
public abstract class NombreClase { ... }
```

2. Métodos abstractos

Un método abstracto es un **método "vacío"**, es decir, no posee cuerpo, por tanto, no puede realizar ninguna acción. La utilidad de un método abstracto es definir qué se debe hacer pero no el cómo se debe hacer.

Los métodos abstractos deben ser implementados por las clases derivadas.

Ejemplo:

```
public abstract class Figura {
  private int numeroLados;
  public Figura() {
     this.numeroLados = 0;
  }
  public abstract float area();  // método abstracto
}

public class Triangulo extends Figura {
  ...
  public float area() { ... }  // Implementación del método (obligatorio)
  ...
}
```

4.2. Sobrescritura (@Override)

1. Sobrescritura de métodos

Un método se puede redefinir (volver a definir con el mismo nombre) en una subclase, para proporcionar una implementación más detallada o adaptada a dicha clase.

Ejemplo:

```
public class Vehiculo {
  public String sVelocidad(int kmh) {
    return "El vehículo circula a: " + kmh + " km/h";
  }
}
```

```
public class Coche extends Vehiculo {
  public String sVelocidad(int kmh) {
    return "El coche circula a: " + kmh + " km/h";
  }
}
```

En este ejemplo, si una aplicación utiliza instancias de la clase Vehículo, también puede trabajar con instancias de Coche, ya que ambas implementaciones del método *sVelocidad()* tienen la misma firma y el mismo tipo de retorno.

2. Etiqueta @Override

La etiqueta @Override indica al compilador la intención de sobrescribir un método.

Si no se escribe esta etiqueta, la sobrescritura del método se realiza igualmente, ya que @Override indica simplemente una intención. Ahora bien, si se utiliza esta etiqueta y el método que se define a continuación no sobrescribe ningún método de la superclase, el compilador avisará con un error.

Ejemplo:

```
public class Coche extends Vehiculo {
  @Override
  public String sVelocidad(int kmh) {
    return "El coche circula a: " + kmh + " km/h";
  }
}
```

```
public class Coche extends Vehiculo {
  @Override
  public String sVelocidades(int kmh) {  // Error
    return "El coche circula a: " + kmh + " km/h";
  }
}
```

3. Vinculación Dinámica

El compilador no puede determinar en tiempo de compilación qué método debe ser invocado, debido a que necesita verificar el tipo de objeto.

Dado que esta verificación ocurre en **tiempo de ejecución**, la vinculación de los métodos sobrescritos se realiza de manera dinámica.

5. Interfaces

A diferencia de otros lenguajes de programación, Java no permite la herencia múltiple, es decir, las clases únicamente pueden heredar de una superclase.

Por tanto, para representar relaciones más complejas Java proporciona un mecanismo denominado interfaz (**interface**).

1. Definición

Una interfaz es una conjunto de **métodos abstractos** y propiedades constantes, donde se especifica qué se debe de hacer pero no cómo, serán las clases que implementen dicha interfaz las que definan el comportamiento.

A diferencia de una clase abstracta, una interfaz no puede hacer (prácticamente) nada por sí sola, es como un "contrato".

Sintaxis:

```
public interface NombreInterfaz {
  public tipo metodo1();
  public tipo metodo2();
  ...
  default public tipo metodoN() { ... }  // Método predeterminado
  ...
}
```

Ejemplo:

```
public interface Mascota {
  public String getCodigo();
}
```

2. Implementación (implements)

Un clase puede implementar una o varias interfaces.

Sintaxis:

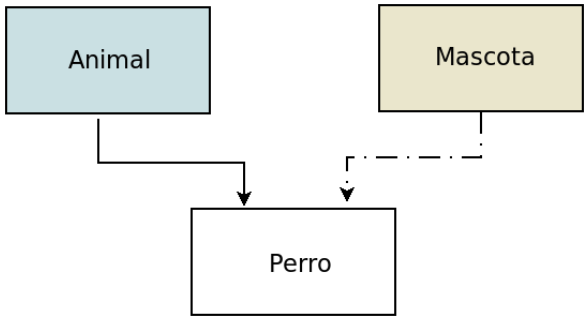
```
public class NombreClase implements NombreInterfaz { ... }
```

Ejemplo:

```
public class Perro implements Mascota {
  ...
  @Override
  public String getCodigo() { ... }  // implementa obligatoriamente el método de la interfaz
  ...
}
```

Además, es posible **combinar** la herencia con el uso de interfaces:

```
public class Perro extends Animal implements Mascota { ... }
```



3. Características

Algunas características de las interfaces son:

- Todos sus miembros son públicos.
- Sus métodos no pueden ser implementados en la interfaz (abstractos), excepto los métodos por defecto.
- Las clases que la implementen deben obligatoriamente implementar TODOS sus métodos, salvo los métodos por defecto.

4. Implementación de múltiples interfaces

Si un clase implementa más de una interfaz, ésta debe implementar todos los métodos de todas interfaces que implemente.

Ejemplo:

```
public interface GPS {
  public void obtenerCoordenadas();
}
```

```
public interface Radio {
  public void iniciarRadio();
  public void detenerRadio();
}
```

```
public class Smartphone implements GPS,Radio {
  @Override
  public void obtenerCoordenadas() {
    // devuelve algunas coordenadas
  }
  @Override
  public void iniciarRadio() {
    // iniciar Radio
  }
  @Override
  public void detenerRadio() {
    // detener Radio
  }
}
```

5. Métodos predeterminados

Los métodos predeterminados se utilizan en el caso de tener que añadir nuevas funcionalidades (métodos) a la interfaz una vez que ésta ya tiene clases que la implementan, de este modo se evita la rotura del código de dichas clases.

Ejemplo:

```
public interface GPS {
  public void obtenerCoordenadas();
  default public void obtenerCoordenadasAproximadas() {
    // implementación para devolver coordenadas de fuentes aproximadas
    // como wifi y móvil
    System.out.println("Obteniendo coordenadas aproximadas por wifi y GSM...");
  }
}
```

```
public class Smartphone implements GPS,Radio { ... }  // no cambia
```

```
// Programa principal
Smartphone pixel6 = new Smartphone();
pixel6.obtenerCoordenadasAproximadas();  // Obteniendo coordenadas aproximadas por wifi y GSM...
```

6. Herencia de interfaces

Una interfaz puede heredar de otra.

Sintaxis:

```
public interface NombreInterfaz extends InterfazPadre { ... }
```

Ejemplo:

```
public interface Reproductor {
  public void iniciar();
  public void pausar();
  public void detener();
}
```

```
public interface ReproductorMusica extends Reproductor {
  public void siguiente();
}
```

```
public class SmartPhone implements ReproductorMusica {
  @Override
  public void iniciar() {
    System.out.println("iniciar");
  }
  @Override
  public void detener() {
    System.out.println("detener");
  }
  @Override
  public void pausar() {
    System.out.println("pausar");
  }
  @Override
  public void siguiente() {
    System.out.println("siguiente");
  }
}
```

6. Modificador "static"

Los métodos y atributos estáticos van precedidos del modificador *static*.

```
static tipo nombreMétodo(parámetros) { }
```

```
static tipo nombreAtributo;
```

1. Métodos estáticos

Un método estático o de clase NO requiere de una instancia (objeto) para ser invocado.

Invocación desde fuera de la clase:

```
NombreClase.métodoEstático(parámetros)
```

Ejemplo:

```
public class PruebaStatic {
  public static void main(String[] args) {
    ClaseConMetodoStatic.saludar("Pepe");  // Salida por pantalla: "Hola Pepe"
  }
}
```

```
class ClaseConMetodoStatic {
  public static void saludar(String nombre) {
    System.out.println("Hola " + nombre);
  }
}
```

1.1. Algunos errores

i. Llamar a un método estático desde una instancia de la clase.

Ejemplo:

```
ClaseConMetodoStatic obj = new ClaseConMetodoStatic();
obj.saludar("Pepe");  // Warning
```

ii. Utilizar desde un método estático miembros no estáticos de la clase.

Ejemplo:

```
class ClaseConMetodoStatic {
  String nombre;
  public static void saludar() {
    System.out.println("Hola " + nombre);  // Error
  }
}
```

2. Atributos estáticos

Un atributo estático o de clase sólo existe una copia para todos los objetos de la clase, y no una para cada objeto. Al igual que los métodos estáticos, NO es necesario instanciar la clase para utilizarlo.

Acceso desde fuera de la clase:

```
NombreClase.atributoEstático  // (Si es público)
```

Estos atributos pueden ser utilizados tanto por métodos estáticos como no estáticos.

6.1. Ejemplo

Clase Coche

```
public class Coche {
  // atributo de clase
  private static int kilometrajeTotal = 0;
  // método de clase
  public static int getKilometrajeTotal() {
    return kilometrajeTotal;
  }
  //Atributos de instancia
  private String marca;
  private String modelo;
  private int kilometraje;
  // Constructor
  public Coche(String marca, String modelo) {
    this.marca = marca;
    this.modelo = modelo;
    this.kilometraje = 0;
  }
  // Get de kilometraje (no estático)
  public int getKilometraje() {
    return kilometraje;
  }
  // Método no estático: utiliza atributos estáticos y no estáticos
  public void recorre(int km) {
    kilometraje += km;
    kilometrajeTotal += km;
  }
}
```

7. Variables y referencias

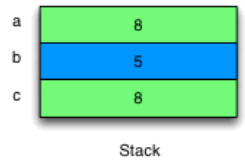
En Java existen dos tipos de variables:

1. Variables primitivas

Son las variables de tipos primitivos (int, float, char, ...), las cuales **almacenan los valores directamente**.

Ejemplo:

```
int a;  // reserva espacio en memoria para almacenar un valor entero (valor por defecto 0)
a = 8;  // almacena el valor 8
int b = 5;  // nueva reserva de espacio, y almacenamiento del valor 5
int c = a;  // nueva reserva de espacio, y copia (duplica) el valor almacenado en la variable "a"
```



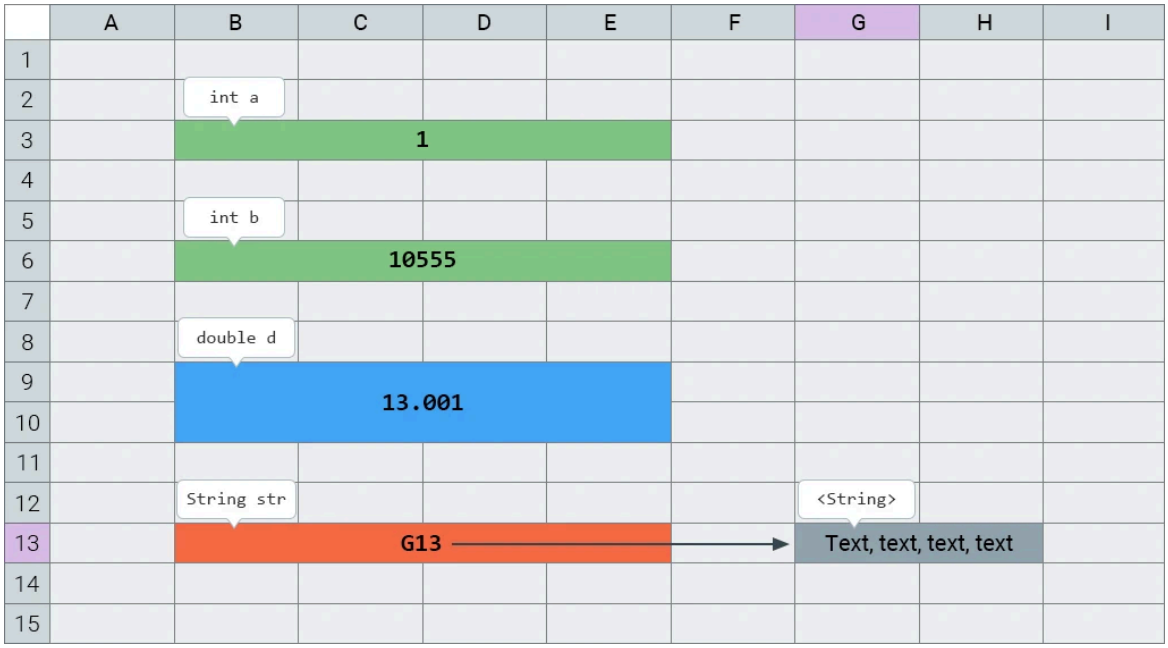
El valor 8 se encuentra almacenado en las variables a y c (duplicado).

2. Variables de referencia

Son las variables de tipos NO primitivos, las cuales **almacenan referencias a objetos** (la dirección de memoria del objeto).

Ejemplo: String

Representación en memoria:



No está permitido modificar las referencias:

```
text++;  // error
```

```
text = 0x1234;  // error
```

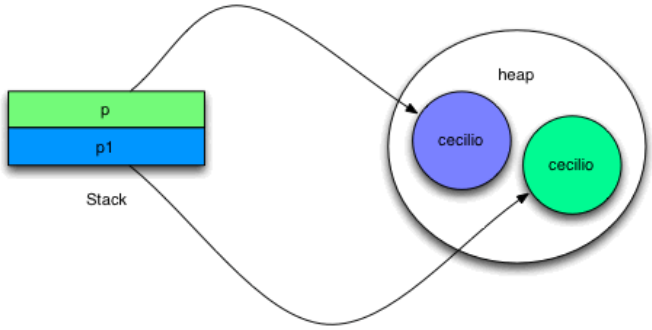
Ejemplo 1:

```
/*
  Variable tipo Cubo que puede almacenar una referencia a un objeto Cubo.
*/
Cubo c1;  // Su valor inicial es null (referencia nula)
```

```
c1 = new Cubo(5);  // asigna la referencia al objeto Cubo creado
```

Ejemplo 2:

```
Persona p  = new Persona("cecilio");
Persona p1 = new Persona("cecilio");
```




```
p == p1  // false, referencias diferentes
```

2.1. Copia de objetos

Al asignar una variable de referencia a otra, **solo se copia la dirección del objeto**, el objeto en sí NO se copia.

Esto evita copiar grandes cantidades de datos (si los objetos son grandes).

Ejemplo 1:

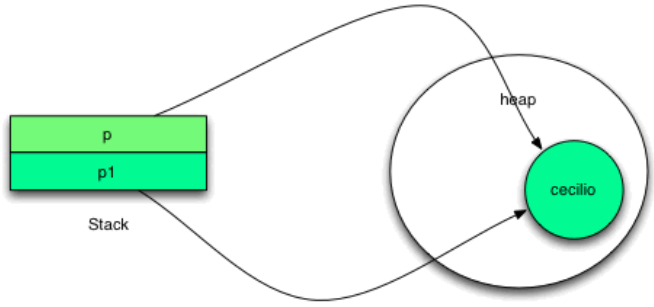
```
Cubo c2 = c1;  // se copia la referencia del objeto
```

```
/*
  Ahora el MISMO objeto Cubo está referenciado por dos variables distintas (c1 y c2).
*/
```

```
c1.setContenido(2);
System.out.println( c2.getContenido() );  // ????
```

Ejemplo 2:

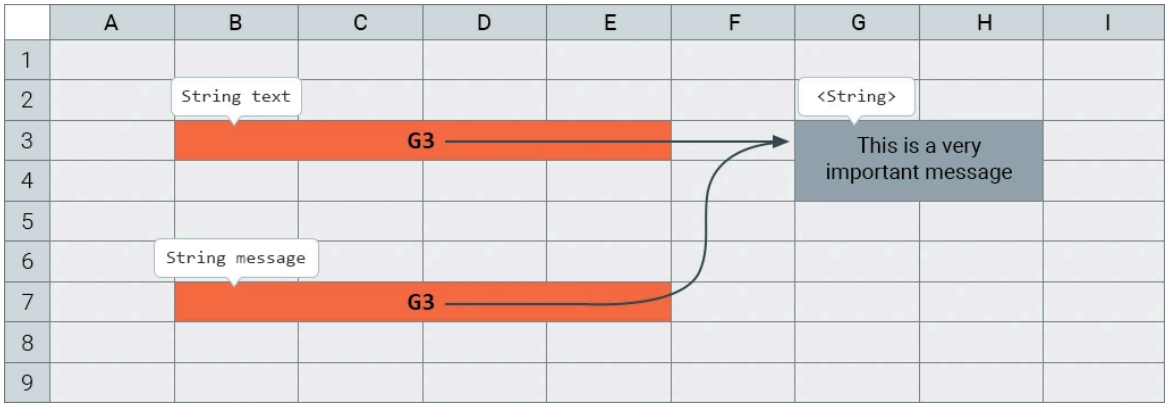
```
Persona p  = new Persona("cecilio");
Persona p1 = p;
```




```
p == p1  // true, es la misma referencia
```

Ejemplo 3:

```
String text = "This is a very important message";
String message = text;
```



Ejemplo 4

```
String text = "This is a very important message";
String message = "This is a very important message";
```

```
text == message  // ¡¡TRUE!! Los literales String apuntan a un mismo objeto (por eficiencia)
```

2.2. Borrado de objetos

Para borrar el contenido de una variable de referencia basta con asignarle el valor null.

Ejemplo:

```
c2 = null;
```

```
/*
  El objeto Cubo aún sigue asociado a (referenciado por) la variable c1.
*/
```

```
c1 = null;  // ¿¿¿Qué ocurre con el objeto Cubo al que apuntaba???
```

RECOLECTOR DE BASURA (*Garbage Collector*)

El GC realiza un reciclado de los objetos de la memoria de la JVM para optimizar el espacio de la misma.

```
System.gc()  // ejecuta de forma inmediata el GC
```

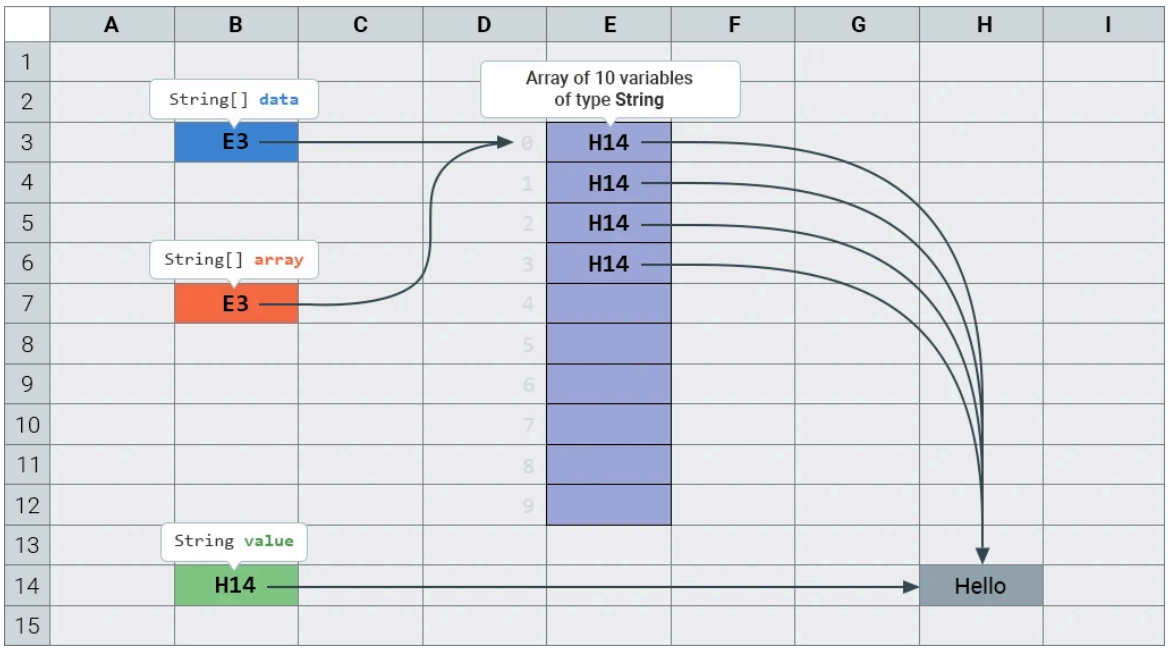
2.3. Objetos como parámetros

Las variables de referencia permiten pasar objetos a métodos como parámetros de forma eficiente, permitiendo además que puedan ser utilizados y modificados (llamando a sus métodos) dentro del propio método.

Esto es debido a que se pasa como parámetro el propio objeto (mediante su referencia), y no una copia del mismo.

Ejemplo:

```
class Ejemplo {
  public static void fill(String[] array, String value) {
    for (int i = 0; i < array.length; i++)
      array[i] = value;
  }
  public static void main(String[] args) {
    String[] data = new String[10];
    fill(data, "Hello");
  }
}
```



3. Diferencias con C++

En C++, una variable puede contener un valor (tipo primitivo), estructura u objeto, o bien ser una referencia a dichos elementos: cuando se pasa un variable a una función o método, se copia su contenido (si se pasa por valor) o no se copia (si se pasa por referencia).

En Java, esta dualidad no existe, los parámetros se pasan a los métodos **siempre por valor**, solo que cuando se trata de variables de referencia, este valor es una referencia.

8. Casting de objetos

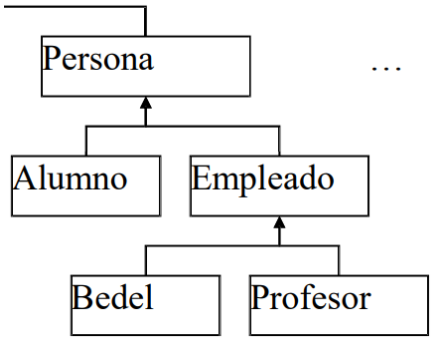
Es posible hacer casting entre objetos de clases relacionadas jerárquicamente (herencia).

1. Tipos

1.1. Casting hacia arriba (upcasting)

Un objeto de la clase derivada siempre se podrá usar en el lugar de un objeto de la clase base (ya que se cumple la relación "es-un"). Este casting se realiza de **forma implícita**.

Ejemplo:




```
Persona p = new Alumno();  // upcasting implícito (un alumno siempre es una persona)
```

O bien:

```
Alumno a1 = new Alumno();
Persona p1 = a1;
```

1.2. Casting hacia abajo (downcasting)

No se produce por defecto, ya que un objeto de la clase base no siempre es un objeto de la clase derivada. En caso de serlo, se tendrán que indicar de **forma explícita**.

Ejemplo:

```
Alumno a2;
a2 = p1;  // error, falta conversión explícita (una persona no siempre es un alumno)
a2 = (Alumno) p1;  // downcasting explícito
```

No es posible realizar este casting si la instancia no es de la clase derivada:

```
Persona p2 = new Persona();
Alumno a3 = (Alumno) p2;  // compila pero lanza un error (excepción) en tiempo de ejecución
```

2. Operador "instanceof"

Comprueba si un objeto es realmente una instancia de una determinada clase o interfaz.

Sintaxis:

```
referenciaObjeto instanceof Clase
```

Ejemplo:

```
(p1 instanceof Alumno)  // true
```

```
(p2 instanceof Alumno)  // false
```

2.1. Más ejemplos

Ejemplo 1:

```
class Animal {}
class Perro extends Animal {}
class Robot {}
```

```
...
```

```
Perro perro = new Perro();
Animal animal = new Animal();
```

```
System.out.println(perro instanceof Animal); // true
System.out.println(perro instanceof Robot);  // false
System.out.println(animal instanceof Perro); // false
```

```
...
```

```
perro = null;
```

```
System.out.println(perro instanceof Animal); // false
```

Ejemplo 2: (con interfaces)

```
interface Vehicle { }
class Car { }
class Ferrari extends Car implements Vehicle { }
```

```
...
```

```
Ferrari ferrari = new Ferrari();
```

```
(ferrari instanceof Ferrari) // true
(ferrari instanceof Vehicle) // true - Ferrari implementa Vehicle
(ferrari instanceof Car)     // true - Ferrari es subclase de Car
(ferrari instanceof Object)  // true - Ferrari es subclase de Object (como todas las clases en Java)
```

```
...
```

```
Car car = new Car();
```

```
(car instanceof Ferrari) // false - Car es superclase de Ferrari, y no es necesariamente una instancia de Ferrari
```

9. Polimorfismo

1. Definición

Capacidad de enviar mensajes similares a objetos de un mismo tipo genérico (superclase o interfaz), y que éstos respondan de manera diferente dependiendo del tipo concreto de objeto que sean.

El polimorfismo permite implementar **programas extensibles**, es decir, se pueden añadir nuevos comportamientos (añadiendo nuevos tipos específicos) sin que varíe su implementación.

2. Comportamiento polimórfico

Para conseguir que objetos se comporten de manera polimórfica (para determinados mensajes), se hará uso de las siguientes características de la POO:

- *Sobrescritura de métodos (¿abstractos?)*
- *Casting de objetos*

dentro de una jerarquía de clases (superclases e interfaces).

Además, será habitual la utilización de *arrays* o *colecciones* de objetos.

3. Ejemplo

Array de cubos

```
Cubo[] cubos = {new CuboPequenio(), new CuboGrande(), new CuboMediano(), new CuboGrande()};
```

```
for (Cubo unCubo : cubos)
  unCubo.logo();  // comportamiento polimórfico
```

10. Ejemplo: clase Cubo

Cubo.java

```
public abstract class Cubo {
  // atributos
  private int capacidad; // capacidad máxima en litros
  private int contenido; // contenido actual en litros
  public static int numCubos = 0;
  public static int capacidadTotal = 0;

  // métodos
  // Método constructor
  Cubo (int c) {
    this.capacidad = c;
    numCubos ++;
    capacidadTotal += this.capacidad;
  }

  // métodos getter
  int getCapacidad() {
    return this.capacidad;
  }
  int getContenido() {
    return this.contenido;
  }
  // método setter
  int setContenido(int litros) {
    int respuesta = 0;

    if (litros > this.capacidad){
        this.contenido = this.capacidad;
        respuesta = 1;
    }
    else if(litros < 0){
        this.contenido = 0;
        respuesta = -1;
    }
```

```
        else{
            this.contenido = litros;
        }
        return respuesta;
    }

    /**
    * Vacía el cubo.
    */
    public void vacia() {
        this.contenido = 0;
    }
    /**
    * Llena el cubo al máximo de su capacidad.
    */
    public void llena() {
        this.contenido = this.capacidad;
    }
    @Override
    public String toString() {
        return "Capacidad: " + this.capacidad + " Contenido: " + this.contenido;
    }

    public abstract void logo();
}
```

CuboPequenio.java

```
public class CuboPequenio extends Cubo{
    public static int totalPequenio = 0;
    public static int totalLitros = 0;

    // Capacidad 5L
    public CuboPequenio() {
        super(5);

        totalPequenio ++;
        totalLitros += this.getCapacidad();
    }
```

```
    @Override
    public void logo(){
        System.out.println("C5L");
    }

}
```

CuboMediano.java

```
public class CuboMediano extends Cubo{

    // Capacidad 10L
    public CuboMediano() {
        super(10);
    }

    // Método estático que muestra un saludo
    public static void saludar(String nombre) {
        System.out.println("Hola " + nombre);
    }

    // Implementa método logo() para CuboMediano
    @Override
    public void logo(){
        System.out.println("C10L");
    }

}
```

PruebaCubo2.java

```
public class PruebaCubo2 {
    public static void main(String[] args) {
        Cubo[] cubos = {new CuboPequenio(), new CuboPequenio(), new CuboMediano(), new CuboPequenio()};
        //Cubo[] cubos2 = {new Cubo(20), new Cubo(30), new Cubo(100), new Cubo(80)};
```

```
        for (Cubo c1 : cubos) {
            c1.setContenido(3);
        }
        for (var c1 : cubos) {
            System.out.println(c1);
            c1.logo(); // Polimorfismo
        }
    }
}
```

Reiniciar tour para usuario en esta página