UT5. CLASES Y OBJETOS

Contenido

1 Paradigma de Programación Orientado a Objetos.

En el paradigma imperativo o procedural clásico, el funcionamiento de una aplicación:

- Está regido por una sucesión de llamadas a diferentes procedimientos y funciones disponibles en el código.
- Estos procedimientos y funciones son los encargados de procesar los datos de la aplicación representados por las variables.
- No se establece relación entre los datos y el código que los manipula.

En el paradigma de Programación Orientado a Objetos, el programa es un conjunto de objetos que interactúan los unos con los otros enviándose mensajes que modifican su estado.

Cuando se ejecuta un programa OO ocurren tres tipos de sucesos:

- Se crean objetos cuando se necesitan.
- Los mensajes se mueven de un objeto a otro, o desde un usuario a un objeto, según se va procesando información, respondiendo a entradas del usuario.
- Se borran los objetos cuando ya no son necesarios.

Pero, ¿qué es un objeto?. Por su interés se presenta a continuación un extracto del tutorial de Java proporcionado por Oracle:

- http://docs.oracle.com/javase/tutorial/java/concepts/object.html

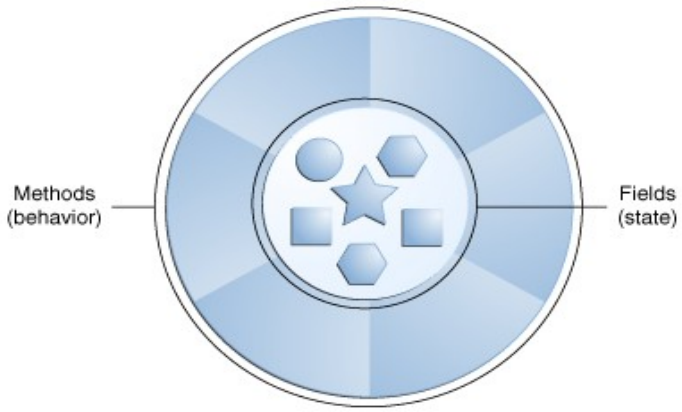
What Is an Object?

Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop

lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.



A software object.

Alan Kay resumió las cinco características básicas de Smalltalk, el primer lenguaje de programación orientado a objetos que tuvo éxito, además de uno de los lenguajes en los que se basa Java. Estas características constituyen un enfoque puro a la programación orientada a objetos:

- **Todo es un objeto.** Piense en cualquier objeto como una variable: almacena datos, permite que se le "hagan peticiones", pidiéndole que desempeñe por sí mismo determinadas operaciones, etc. En teoría, puede acogerse cualquier componente conceptual del problema a resolver (bien sean perros, edificios, servicios, etc.) y representarlos como objetos dentro de un programa.
- **Un programa es un cúmulo de objetos** que se dicen entre sí lo que tienen que hacer mediante el envío de mensajes. Para hacer una petición a un objeto, basta con "enviarle un mensaje". Más concretamente, puede considerarse que un mensaje en sí es una petición para solicitar una llamada a una función que pertenece a un objeto en particular.
- **Cada objeto tiene su propia memoria**, constituida por otros objetos. Dicho de otra manera,no crea una nueva clase de objeto construyendo un paquete que contiene objetos ya existentes. Por consiguiente, uno puede incrementar la complejidad de un programa, ocultándola tras la simplicidad de los propios objetos.
- **Todo objeto es de algún tipo.** Cada objeto es un elemento de una clase, entendiendo por "clase" un sinónimo de "tipo". La característica más

relevante de una clase la constituyen "el conjunto de mensajes que se le pueden enviar".

- **Todos los objetos de determinado tipo pueden recibir los mismos mensajes.** Ésta es una afirmación de enorme trascendencia como se verá más tarde. Dado que un objeto de tipo "círculo" es también un objeto de tipo "polígono" (de infinitos lados), se garantiza que todos los objetos "círculo" acepten mensajes propios de "polígono". Esto permite la escritura de código que haga referencia a polígonos, y que de manera automática pueda manejar cualquier elemento que encaje con la descripción de "polígono". Esta capacidad de suplantación es uno de los conceptos más potentes de la POO.

En la programación orientada a objetos hay tres conceptos son fundamentales:

- La encapsulación
- La herencia
- El polimorfismo

Se verán más adelante.

2 Clases: Atributos. Métodos.

Una clase es la unidad fundamental en programación, la pieza de Lego. El problema es... que no existe. Es una abstracción. Es la plantilla que utilizaremos posteriormente para crear un conjunto de objetos con características similares.

Las clases agrupan en una sola estructura:

- Los datos que nos interesan sobre algo (atributos)
- Las operaciones que se pueden realizar sobre dichos datos (métodos)

Supongamos que definimos una clase Coche. No tiene entidad física. Hablamos de un coche en general, sin especificar de qué tipo de coche se trata. Podemos asignarle un comportamiento y una serie de características. A partir de esa clase Coche, podremos crear nuestros objetos (también llamados instancias), que serán las realizaciones ”físicas” de la clase. En el ejemplo, se muestran un Seat Panda, un Opel Corsa, y un Renault Megane. Todos ellos comparten una serie de características comunes por las que podemos identificarlos como coches.

Vemos en el código que, respecto al comportamiento, podemos arrancar y detener nuestro coche. Esos son los métodos de la clase. En cuanto a las características (llamadas atributos), tenemos las variables estadoMotor, color, y modelo. Para definir nuestra clase en Java, lo haremos de la siguiente manera:

La clase Coche.

```
class Coche {
	boolean estadoMotor = false;
	String color;
	String modelo;
	void arrancar(){
		if (estadoMotor == true)
			System.out.println("El coche ya est´a arrancado");
		else{
			estadoMotor=true;
			System.out.println("Coche arrancado");

		}
	}
	void detener(){
		if(estadoMotor == false)
			System.out.println("El coche ya está detenido");
		else{
			estadoMotor=false;
			System.out.println("Coche detenido");
		}
	}
} // fin de la clase Coche
```

Podemos escribir la clase anterior y guardarla en un archivo con el nombre Coche.java, por ejemplo. Como ya se explicó en la introducción, todos los archivos con código fuente de Java llevan la extensión java.

Si lo compilamos (javac Coche.java) obtendremos un archivo binario llamado Coche.class (nótese que, si hubiéramos definido varias clases en el mismo archivo, el compilador nos crearía tantos binarios con extensión class como clases hubiéramos definido). Si, a continuación, intentamos ejecutar esa clase (java Coche), nos dará un error, informándonos de que no tenemos creado un método main.

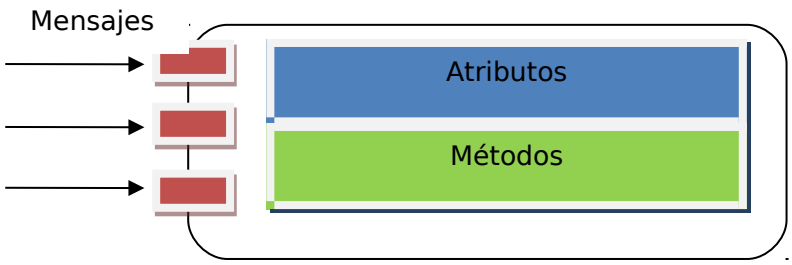
Es decir, que una clase, por sí sola, no puede ejecutarse. O bien definimos un método main que nos cree un objeto de esa clase con el que trabajar, o bien llamamos a un objeto de esa clase desde otro objeto perteneciente a otra clase.

3 Objetos: Estado y Comportamiento. Mensajes

Un ***objeto*** (en inglés, ***instance***) es un ejemplar concreto de una clase, es decir, una instancia de una clase. Los objetos se crean a partir de las clases.

Los objetos tienen un estado definido por sus variables, y un comportamiento, determinado por sus métodos.

- **Estado:** son los valores que toma un objeto para cada uno de sus atributos.
- **Comportamiento:** viene determinado por los métodos que forman la clase, que son los que fijan las operaciones que pueden realizar un objeto.



Representación gráfica de un objeto

- **Mensajes:** Un mensaje es una llamada a un método de una clase.

Una clase, por sí sola, no puede ejecutarse. O bien definimos un método main que nos cree un objeto de esa clase, con el que podremos trabajar, o bien llamamos a un objeto de esa clase desde otro objeto perteneciente a otra clase.

Vamos a coger el ejemplo de la clase Coche, propuesto, y vamos a definir un método main donde podamos instanciar la clase:

```
class Coche {

	boolean estadoMotor = false;
	String color;
	String modelo;
```

```
	void arrancar(){
		if(estadoMotor == true)
			System.out.println("El coche ya est´a
		arrancado");
		else{
			estadoMotor=true;
			System.out.println("Coche arrancado");
		}
	}
	void detener(){
		if(estadoMotor == false)
			System.out.println("El coche ya est´a
		detenido");
		else{
```

```
                estadoMotor=false;
                System.out.println("Coche detenido");
            }
        }
        public static void main(String args[]){
            Coche ferrari = new Coche();
            ferrari.color="rojo";
            ferrari.modelo="Diablo";

            System.out.println("El modelo del coche es " +
        ferrari.modelo + " y es de color " + ferrari.color);
            System.out.println("Intentando detener el coche...");
            ferrari.detener();
            System.out.println("Intentando arrancar el
        coche...");
            ferrari.arrancar();
        }
} // fin de la clase Coche.
```

Carácterísticas de los objetos

- Se agrupan en clases.
- Contienen datos internos que definen su estado.
- Permiten ocultar los datos que contienen (encapsulación)
- Pueden heredar propiedades de otros objetos. (herencia)
- Pueden comunicarse con otros objetos, enviando y recibiendo mensajes.
- Tienes métodos que definen su comportamiento.

4 Instanciación de Objetos: Declaración y Creación

En primer lugar, hay que aclarar el concepto de *instancia*. Una instancia es un objeto. Por tanto, cuando hablemos de *instanciar una clase*, nos estaremos refiriendo al proceso de crear un objeto a partir de esa clase.

La instanciación de un objeto consiste en asignar memoria al objeto para que pueda guardar los datos. Para poder realizar esto hay que llevar a cabo estos dos pasos:

1.- Declarar el objeto

```
Coche ferrari
```

2.- Crear el objeto

Para crear un objeto utilizaremos el operador new con el nombre de la clase que se desea instanciar.

```
            System.out.println("Intentando arrancar el coche...");
            ferrari.arrancar();
        }
} // fin de la clase Coche.
```

Aquí es donde se crea el objeto propiamente dicho, es decir, donde se reserva memoria para guardar los datos de dicho objeto.

```
ferrari = new Coche();
```

Podemos agrupar la declaración y creación de la siguiente manera:

```
Coche ferrari = new Coche();
```

Lo que estamos haciendo realmente al crear el objeto es llamar a su constructor. El constructor no es más que un método que inicializa los parámetros que pueda necesitar la clase. En secciones posteriores se verá con detalle. Por ahora, sólo nos interesa el hecho de que la forma de crear una instancia es llamando al constructor de la clase.

Hemos creado un objeto, pero ¿qué pasa cuando ya no lo necesitemos más?, ¿existe alguna forma de destruirlo?. La respuesta es que sí es posible, pero no hace falta. En Java, la administración de memoria es dinámica y automática. Existe un recolector de basura que se ocupa, continuamente, de buscar objetos no utilizados y borrarlos, liberando memoria en el ordenador.

5 Utilización de Atributos

Las propiedades son los datos del objeto.

Para realizar la declaración de atributos se debe indicar el ámbito (package, public, protected, private), tipo (int, double...) e identificador.

Ejemplo:

private int numero;

Un ejemplo de utilización se presenta a continuación:

```
class Coche {

     boolean estadoMotor = false;
     String color;
     String modelo;

     void arrancar(){
          if(estadoMotor == true)
               System.out.println("El coche ya est´a
          arrancado");
          else{
               estadoMotor=true;
               System.out.println("Coche arrancado");
```

```
        }
    }

    void detener(){
        if(estadoMotor == false)
            System.out.println("El coche ya est´a
        detenido");
        else{
            estadoMotor=false;
            System.out.println("Coche detenido");
        }
    }
    public static void main(String args[]){
        Coche ferrari = new Coche();
        ferrari.color="rojo";
        ferrari.modelo="Diablo";

        System.out.println("El modelo del coche es " +
    ferrari.modelo + " y es de color " + ferrari.color);
        System.out.println("Intentando detener el coche...");
        ferrari.detener();
        System.out.println("Intentando arrancar el
    coche...");
        ferrari.arrancar();
    }
} // fin de la clase Coche.
```

Para poder utilizar las propiedades de un objeto fuera de la clase debemos poner: *nombre_del_objeto.nombre de la propiedad*.

Ejemplo: *ferrari.color*

NOTA: Siempre y cuando la propiedad sea visible desde la clase llamante. La visibilidad se verá más adelante.

Para poder hacer uso de las propiedades de la clase dentro de la propia clase, simplemente se hace indicando el nombre de dicha propiedad.

NOTA: También se puede utilizar la palabra clave this. Lo veremos más adelante.

6 Utilización de Métodos: Parámetros y Valores de Retorno.

Los métodos nos definen el comportamiento (la funcionalidad) de un objeto.

Un método consta de las siguientes partes:

- Uno o varios modificadores (public, static, final, etc.).
- El objeto o tipo primitivo que devuelve.

- El nombre del método.
- Un conjunto de parámetros.
- La palabra reservada throws, que permite arrojar excepciones, y que se verá en otro tema más adelante.
- El cuerpo del método.

Un ejemplo de método:

```
public int saludo(String nombre){
      int hayError=0; //Si 0,no error

      if(nombre==null)
            return (-1);
      else
      {
            System.out.println("Hola, "+nombre+", la prueba ha
      sido correcta");
            return 0;
      }
}
```

En el listado anterior, public es el modificador, int el tipo primitivo que devuelve el método, saludo el nombre del método, nombre el parámetro que se introduce, y todo lo que va entre las llaves del método es el *cuerpo del método*. Es un método muy sencillo, que se limita a mostrar por pantalla un saludo a la persona cuyo nombre hemos introducido como parámetro. Los valores que devuelve son: 0, si no ha habido ningún problema, o -1 en caso de que no se haya introducido ningún nombre. Nótese que los valores se devuelven mediante la palabra reservada return. Evidentemente, el tipo u objeto devuelto por return debe coincidir con el tipo de valor de vuelta especificado en el método (int, en este ejemplo) o el compilador nos dará un error.

Cuando no se devuelva nada en un método, debe especificarse el tipo de vuelta como void. Por ejemplo: public static void main(String args[]).

Un inciso: si no se introduce un nombre, el objeto String nombre estará vacío. Para comprobar si un objeto está vacío, utilizamos la palabra reservada null. Todo objeto que no esté inicializado estará a null.

Cuando se pasan parámetros, debe tenerse en cuenta que aunque el paso de parámetros en Java es por valor, en el caso de los datos primitivos y de los objetos se maneja de diferente manera.

- En el caso de los tipos primitivos se crea una copia dentro del método, y es con esa copia con la que se trabaja. El tipo original mantiene el valor que tenía.

- Para los objetos se pasa una referencia (de la que se hace copia). Como lo que estamos pasando es una referencia (un apuntador) al objeto, todo lo que hagamos a ese objeto dentro del método afectará al objeto original. En este caso se encuentran los arrays y a los objetos que contienen.

7 Constructores

Un constructor es el método que implementa las acciones necesarias para inicializar la instancia de la clase, y es invocado por el operador *new* cuando instanciamos dicha clase.

Por lo tanto, cuando desarrollamos una clase, el o los constructores deben tener la capacidad de inicializar los aspectos básicos de funcionamiento del objeto.

Por tanto, la utilidad del constructor es dar valores iniciales a los datos (propiedades) del objeto en el mismo momento en el que se reserva memoria.

Si para una clase no se define ningún método constructor, Java ejecuta su función constructora por defecto, que tiene la siguiente cabecera: public nombreClase(). Lo que hace es dar a las variables numéricas el valor 0 y a las de tipo referencia el valor null.

Si la función constructora creada por el programador no da valores iniciales a todos los atributos de la clase, Java inicializa dichos atributos con las valores indicados anteriormente.

Si una clase tiene definida una función constructora, esto impedirá que se ejecute la función constructora por defecto que tiene Java.

He aquí hay una clase con un constructor sin argumentos:

```
//: c04:ConstructorSimple.java
// Muestra de un constructor simple.
class Roca {
     Roca() { // Éste es el constructor
     System.out.println("Creando Roca");
     }
}


public class ConstructorSimple {
     public static void main (String[] args) {
          for(int i = 0; i < 10; i++)
               new Roca () ;
     }
}
```

Ahora, al crear un objeto

```
new Roca ();
```

se asigna almacenamiento y se invoca al constructor. Queda garantizado que el objeto será inicializado de manera adecuada antes de poder poner las manos sobre él. Fíjate que el estilo de codificación de hacer que la primera letra de todos los métodos sea minúscula no se aplica a los constructores, dado que el nombre del constructor debe coincidir ***exactamente*** con el nombre de la clase.

Como cualquier método, el constructor puede tener parámetros para permitir especificar ***cómo*** se crea un objeto. El ejemplo de arriba puede cambiarse sencillamente de forma que el constructor reciba un argumento:

```
//: c04:ConstructorSimple2.java
// Los constructores pueden tener parámetros.
class Roca2 {
     Roca2 (int i) {
          System.out.println( "Creando la roca numero " + i) ;
     }
}
public class ContructorSimple2 {
     public static void main(String[l args) {
          for (int i = O; i < 10; i++)
               new Roca2 (i);
     }
}
```

Los constructores eliminan un montón de problemas y simplifican la lectura del código. En el fragmento de código anterior, por ejemplo, no se verá ninguna llamada explícita a ningún método **inicializar(**) que esté conceptualmente separado, por definición. En Java, la definición e inicialización son conceptos que están unidos -no se puede tener uno sin el otro.

El constructor es un tipo inusual de método porque no tiene valor de retorno. Esto es muy diferente al valor de retorno **void,** en el que el método no devuelve nada pero se sigue teniendo la opción de hacer que devuelva algo más. Los constructores no devuelven nada y no es necesario tener ninguna opción. Si hubiera un valor de retorno, y si se pudiera seleccionar el propio, el compilador, de alguna manera, necesitaría saber qué hacer con ese valor de retorno.

8 Referencia a la palabra clave this.

Cada objeto puede acceder a una referencia explícita a sí mismo, sus métodos y atributos mediante la palabra clave *this* (también conocida como referencia *this*).

Ejemplo:

```
public class Persona {

   private String nombre;
   private String apellidos;

   public Persona(String param_nombre, String param_apellidos) {
      this.nombre = param_nombre;
      this.apellidos = param_apellidos;
   }
}
```

La palabra clave *this* no sería necesaria en el caso anterior, ya que el cuerpo del método utiliza en forma implícita la palabra clave *this* para hacer referencia a los atributos y los demás métodos del objeto.

```
public class Persona {

   private String nombre;
   private String apellidos;

   public Persona(String paramnombre, String param_apellidos) {
      nombre = param_nombre;
      apellidos = param_apellidos;
   }
}
```

Sin embargo, a menudo se produce un error lógico cuando un método contiene un parámetro o variable local con el mismo nombre que un atributo de la clase. En tal caso, debe usar referencia *this* si desea acceder al atributo de la clase; de no ser así, se hará referencia al parámetro o variable local del método.

```
public class Persona {

	private String nombre;
	private String apellidos;

	public Persona(String nombre, String apellidos) {
		this.nombre = nombre;
		this.apellidos = apellidos;
	}
 }
```

9 Visibilidad

Los modificadores para determinar su visibilidad son:

package:

- `package`: Si a un mimbre de una clase o a una clase no se le asigna ningún modificador de acceso específico, toma el nivel de acceso por defecto que es conocido como *package*. En este caso, la visibilidad se limitará al paquete donde está definida la clase.
- `public`: Si a un miembro de una clase se el asigna el modificador *public* significa que son accesibles desde cualquier otra clase.
- `protected`: Si a un miembro de una clase se le asigna el modificador indica que se puede utilizar el método en
  - la clase donde está definido,
  - en las subclases de esta clase y
  - en las otras clases que forman parte del mismo paquete.
  - *Las clases nunca podrán ser protected. Es un modificador de visibilidad para los atributos y métodos de clase.*
- `private`: Si a un miembro de una clase se le asigna el modificador *private* indica que el método, sólo puede ser usado en la clase donde está definido.

  Las clases nunca podrán ser private. Es un modificador de visibilidad para los atributos y métodos de clase.

10 Encapsulación.

La encapsulación consiste en agrupar los métodos y los atributos de la clase de tal forma que el objeto presente un interfaz al usuario en el que los detalles de

implementación internos sean transparentes al programador que utiliza la clase. Esto permite al diseñador de la clase cambiar el funcionamiento interno de la clase sin que este afecte a las aplicaciones que la utilizan.

En la práctica implica este concepto consiste en la ocultación del estado o de los datos miembro de un objeto, de forma que sólo es posible modificar los mismos mediante los métodos definidos para dicho objeto.

Poniendo un ejemplo de la industria de la automóvil, ¿conoce usted cómo funciona la caja de cambios de su vehículo?

Para cambiar de marcha, ¿modificará directamente la posición de los engranajes? Afortunadamente, no. Los constructores tienen previstas soluciones más prácticas para la manipulación de la caja de cambios.

Los elementos visibles de una clase desde su exterior son denominados ***interface de clase***. *En el caso de nuestro coche, la palanca de cambios constituye la interface de la caja de cambios. Es a través de ella que podrá actuar sin riesgo sobre los mecanismos internos de la caja de cambios.*

11 Métodos Get y Set

Para favorecer la encapsulación una de las soluciones es ocultar los campos de la clase de tal forma que solo permitan la manipulación mediante *métodos* de esa clase.

Una manipulación típica podría ser el ajuste del saldo bancario de un cliente (por ejemplo, una variable de instancia private de una clase llamada CuentaBancaria) mediante un método llamado calcularInteres.

Las clases a menudo proporcionan métodos public para permitir a los clientes de la clase establecer – set- (es decir, asignar valores) u obtener – get- (es decir, recibir los valores de) atributos ocultos.

El uso de métodos get y set permite que las clases sean más fáciles de depurar y mantener, por lo que deben utilizarse siempre que sea necesario para acceder a los campos de las clases ocultas. Es decir, favorece la encapsulación.

Ejemplo:

```
public class Persona {
   private String nombre;
   private String apellidos;
   public String getNombre() {
      return nombre;
   }
   public void setNombre(String nombre) {
      this.nombre = nombre;
```

```
    }
    public String getApellidos() {
        return apellidos;
    }
    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
    public Persona(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }

}
```

12 Sobrecarga de métodos

Uno de los aspectos más importantes de cualquier lenguaje de programación es el uso de los nombres. Al crear un objeto, se da un nombre a cierta región de almacenamiento. Un método es un nombre que se asigna a una acción. Al utilizar nombres para describir el sistema, se crea un programa más fácil de entender y de modificar por la gente. Es como escribir en prosa -la meta es comunicarse con los lectores.

Para hacer referencia a objetos y métodos se usan nombres. Los nombres bien elegidos hacen más sencillo que todos entiendan un código.

Surge un problema cuando se trata de establecer una correspondencia entre el concepto de matiz del lenguaje humano y un lenguaje de programación. A menudo, la misma palabra expresa varios significados -se ha ***sobrecargado.*** Esto es útil, especialmente cuando incluye diferencias triviales. Se dice "lava la camisa", "lava el coche" y "lava el perro". Sería estúpido tener que decir "lavaCamisas la camisa", "lavacoche el coche" y "lavaperro el perro" simplemente para que el que lo escuche no tenga necesidad de intentar distinguir entre las acciones que se llevan a cabo. La mayoría de los lenguajes humanos son redundantes, por lo que incluso aunque se te olviden unas pocas palabras, se sigue pudiendo entender. No son necesarios identificadores únicos -se puede deducir el significado del contexto.

En Java (y C++) otros factores fuerzan la sobrecarga de los nombres de método: el constructor.

Dado que el nombre del constructor está predeterminado por el nombre de la clase, sólo puede haber un nombre de constructor. Pero ¿qué ocurre si se desea crear un objeto de más de una manera?

Por ejemplo, suponga que se construye una clase que puede inicializarse a sí misma de manera estándar o leyendo información de un archivo. Se necesitan dos constructores, uno que no tome argumentos (el constructor por defecto, llamado también constructor sin parámetros), y otro que tome como parámetro un **String,** que es el nombre del archivo con el cual inicializar el objeto. Ambos son constructores, por lo que deben tener el mismo nombre -el nombre de la clase. Por consiguiente, la sobrecarga de métodos es esencial para permitir que se use el mismo nombre de métodos con distintos tipos de parámetros. Y aunque la sobrecarga de métodos es una necesidad para los constructores, es bastante conveniente y se puede usar con cualquier método.

He aquí un ejemplo que muestra métodos sobrecargados, tanto constructores como ordinarios:

```
//: c04:Sobrecarga.java
// Muestra de sobrecarga de métodos
// tanto constructores como ordinarios.
import java.util.*;

class Arbol {
     int altura;

     Arbol () {
          visualizar ("Plantando un retoño") ;
          altura = 0;
     }

     Arbol (int i) {
          visualizar("Creando un nuevo arbol que tiene " + i +
     metros de alto");
          altura = i;
     }

     void info () {
          visualizar("E1 arbol tiene " + altura + "" metros de
     alto");
     }

     void info(String S) {
          visualizar(s + ": El arbol tiene " + altura + "
     metros de alto");
     }
```

```
    static void visualizar (String S) {
        System.out.println (S) ;
    }

    public class sobrecarga {
        public static void main (String[] args) {
            for(int i = O; i < 5; i++) {
                Arbol t = new Arbol (i) ;
            t.info();
            t. info ("metodo sobrecargado");
        }
        // Constructor sobrecargado:
        new Arbol() ;
    }
}
```

Se puede crear un objeto **Arbol,** bien como un retoño, sin argumentos, o como una planta que crece en un criadero, con una altura ya existente. Para dar soporte a esto, hay dos constructores, uno que no toma argumentos (a los constructores sin argumentos se les llama constructores por defecto1), y uno que toma la altura existente.

Podríamos también querer invocar al método **info(**) de más de una manera. Por ejemplo, con un parámetro **String** si se tiene un mensaje extra para imprimir, y sin él si no se tiene nada más que decir. Parecería extraño dar dos nombres separados a lo que es obviamente el mismo concepto.

Afortunadamente, la sobrecarga de métodos permite usar el mismo nombre para ambos.

13 Almacenamiento en Memoria. Tipos Básicos vs Objetos

Cuando declaramos una variable hay que indicar el tipo de dicha variable. Gracias a este tipo podemos determinar la forma en que se almacena el dato en memoria:

- En el caso de que la variable sea de un tipo básico del lenguaje (int, float, ...), en el momento en el que se declara la variable se reserva un espacio de memoria.
- Si la variable es de objeto, en el momento en que se realiza la declaración no se reserva memoria para guardar los datos. Es necesario que se llame al constructor (new) para que se realice la asignación de dicho espacio; o asignarle la dirección de otro objeto en memoria.

14 Destrucción de Objetos y Liberación de Memoria

El operador new se utiliza para reservar memoria dinámicamente para la creación de los objetos.

Sin embargo, la destrucción de los objetos, y la consiguiente liberación de su memoria utilizada, se realiza en Java de manera automática. A este mecanismo se le denomina *recogida de basura* (garbage collection), su funcionamiento es el siguiente: cuando no hay ninguna referencia a un objeto determinado, se asume que ese objeto no se va a utilizar más, y la memoria ocupada por el objeto se libera, este proceso es ejecutado por un thread de baja prioridad.

No hay necesidad, por tanto, en Java, de destruir explícitamente los objetos, como sucede en C++.

En Java no se sabe exactamente cuándo se va a activar el garbage collector. Se puede llamar explícitamente al garbage collector con el método System.gc().

Cuando el GC libera un espacio de memoria se ejecuta automáticamente el método protected void finalice() siempre que dicho método esté implementado dentro de la clase a la que pertenece el espacio de memoria que se está liberando.