Complementos de Programación

Tema 2. Clases

Andrés Cano Utrera
Departamento de Ciencias de la Computación e I.A.

ETSIIT Universidad de Granada

Curso 2015-16

Índice I

Índice II

Contenido del tema

Motivación

Objetivo del tema

Comprender con más detalle cómo definir y usar **clases**: herramienta fundamental para la programación orientada a objetos.

Motivación (II)

Los conceptos básicos a comprender son:

- Clase y objeto
- Métodos como operaciones sobre objetos
- Encapsulamiento de datos y métodos: abstracción
- Diferencia entre creación y uso de objetos
- Control de acceso
- Interacciones entre objetos
- Uso de notación UML

Contenido del tema

Clase

Definición

Plantilla que define los datos (***estado***) que contienen los objetos de esa clase y los métodos (***comportamiento***) que operan sobre esos datos.

- La clase es la **descripción *general*** de una entidad sobre la que estamos interesados en realizar algún tipo de procesamiento informático.
- Ejemplos: Persona, Coche, Libro, Alumno, Producto, Número complejo, Grafo, Sudoku, ...
- La clase es la base de la PDO en Java.

Clase

```java

// Clase Animal: permite representar a los animales
class Animal{
   // Datos que caracterizan a todos los animales
   int numeroPatas;
   int numeroOjos;
   boolean plumas; // Posibles valores verdadero/false
   int edadActual;
   int longevidad;

   // Metodos miembro

   // Metodo para calcular de forma teorica el tiempo
   // de vida restante de un animal
   int calcularTiempoRestante(){
      return longevidad-edadActual;
   }

   // Metodo para presentar por pantalla la dieta del
   // animal
   void presentarDieta(){
      //.............
   }

   // Resto de metodos de la clase
   //.....................
}
```

Clase

```java
// Clase para representar a los deportistas
class Deportista{
   // Datos miembros
   int edad;
   String ciudad;
   String nombre;

   // Metodos miembro

   // Metodo para imprimir los datos de un deportista
   void imprimirDatos(){
      System.out.println("Nombre: "+nombre);
      System.out.println("Ciudad: "+ciudad);
   }

   // Resto de metodos de la clase
   //.....................
}
```

Nota:

- ¿Tendría sentido incluir un dato miembro longitudCarrera?: nos alejamos de la generalidad.
- Las clases anteriores no tienen método `main`, con lo que no pueden ser ejecutadas. Contienen simplemente las plantillas para construir objetos de esa clase.

Contenido del tema

Objeto

Definición

Instancia de una clase: descripción concreta de un elemento de una clase.

- El ***estado*** de un objeto está representado por los datos de su clase con sus valores concretos.
- El ***comportamiento*** (también llamado ***acciones*** o ***mensajes***) viene definido por los métodos de su clase.

Ejemplo

Clases: Persona, Coche, Libro, Números complejo, ...

Objetos: Pepe Pérez, 3216-BXP, Quijote, $2 + 3i$, ...

Objeto

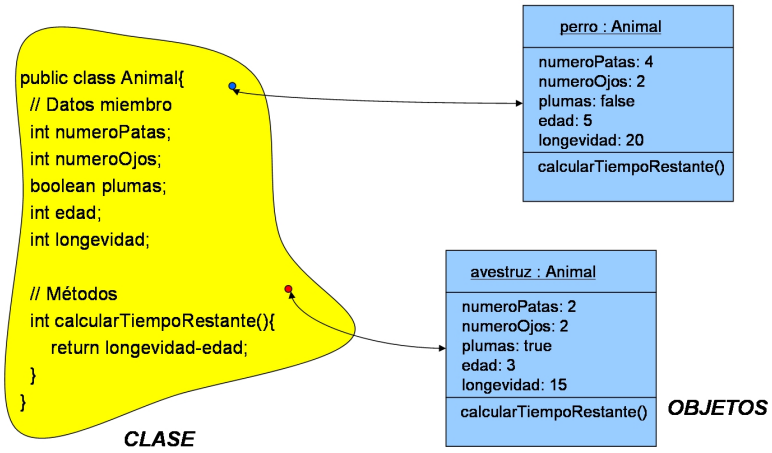
NOTA

- Puede haber (suele haber) varios objetos de cada clase
- Cada objeto pertenece a una clase (por ahora...)

Relación entre clase y objeto

- Clase: conceptualización (datos y métodos) de algún tipo de entidad
- Objeto: realización específica de la entidad (los datos tendrán valores concretos y las operaciones actuarán sobre ellos)

Objeto



Contenido del tema

Creación de objetos y referencias

Importante

- La creación de una clase NO implica la creación automática de objetos
- Los objetos deben crearse de forma explícita
- Símil: clase como planos de construcción de avión. A partir de los planos se pueden construir objetos, pero los planos NO son objetos
- Creación de objeto: reserva de espacio de memoria
- Durante la ejecución del programa puede haber varios objetos de la misma clase (y de otras) en memoria

Creación de objetos y referencias

Ejemplo:

Empresa de empaquetado precisa programa de gestión de cajas:

- Clase: Caja
- Datos miembro: largo, ancho, alto
- Métodos miembro: calcularVolumen

Creación de objetos y referencias

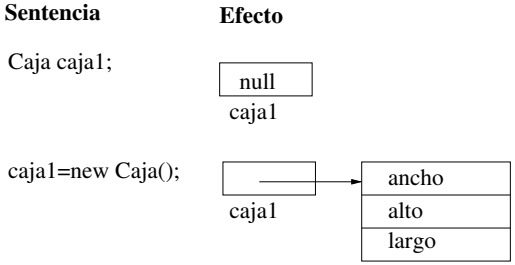
```
// Clase para representar cajas
public class Caja{
  // Datos miembros;
  int alto;
  int ancho;
  int largo;

  // Metodos
  int calcularVolumen(){
    return (alto*ancho*largo);
  }
}
```


Creación de objetos y referencias

Creación de objetos de una clase

- Declarar una variable (***referencia***) del tipo de la clase.
- Obtener una copia física y real del objeto con el operador **new** asignándosela a la variable.

```
Caja caja; // Declaracion de variable referencia
caja = new Caja(); // Creacion del objeto
o bien
Caja caja = new Caja();
```



Creación de objetos y referencias

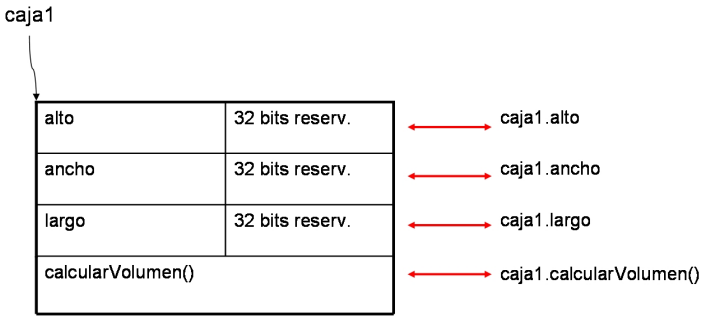
```
// Clase para representar cajas
class Caja{
  // Datos miembro
  int alto;
  int ancho;
  int largo;
  // Metodos
  int calcularVolumen(){
    return alto*ancho*largo;
  }
}
class TestCaja{
  public static void main(String args[]){
    Caja caja1=new Caja();
    caja1.alto=2;
    caja1.ancho=2;
    caja1.largo=10;
    int vol=caja1.calcularVolumen();
    System.out.println("Volumen: "+vol);
  }
}
```

Creación de objetos y referencias

Operador new

- Reserva memoria dinámicamente para un objeto.
- Y hace que se llame al constructor de la clase. En el caso anterior, se llama al *constructor por defecto*.

La plantilla creada permite acceder a sus elementos:



Valor por defecto de los datos de una clase

Valor null

Cuando no se ha asignado ningún valor a una variable referencia, ésta contiene el valor `null`

Valor por defecto de los datos de una clase

- Variables referencia y arrays: `null`
- Tipos numéricos: `0`
- Tipos `boolean`: `false`
- Tipo `char`: `\u0000`

Valor por defecto de los datos de una clase

```
class Estudiante {
   String nombre; // por defecto vale null
   int edad; // por defecto vale 0
   boolean esAlumnoCiencias; //  por defecto vale false
   char genero; //  por defecto vale  '\u0000'
}
class Test {
   public static void main(String[] args) {
      Estudiante estudiante = new Estudiante();
      System.out.println("nombre? " + estudiante.nombre );
      System.out.println("edad? " + estudiante.edad );
      System.out.println("es alumno de Ciencias? " +
                         estudiante.esAlumnoCiencias );
      System.out.println("genero? " + estudiante.genero );
   }
}
```

Salida del programa

```
nombre? null
edad? 0
es alumno de Ciencias? false
genero?
```


Creación de objetos y referencias

Ejercicio:

Analizad el código siguiente y detectad los posibles problemas que pudiera contener (si es que contiene alguno):

Creación de objetos y referencias

```
// Clase para representar cajas
class Caja{
  // Datos miembro
  int alto,ancho,largo;

  // Metodos
  int calcularVolumen(){
    int resultado=alto*ancho*largo
  }
}
class TestCaja{
  // Metodo principal
  public static void main(String args[]){
    Caja caja1=new CajaSimple();
    Caja caja2=new CajaSimple();
    caja1.alto=23;
    caja1.ancho=2;
    caja1.largo=10;
    caja2.ancho=23.4;
    caja2.alto=3.98;
    // Se calcula el volumen de la segunda caja
    int volumen=caja2.calcularVolumen();
    System.out.println("Volumen: "+volumen);
  }
}
```


Creación de objetos y referencias

Problemas:

- ¿acaban las sentencias en punto y coma?
- ¿qué se precisa para conseguir que el resultado de un cálculo en un método sea devuelto al método llamante?
- ¿hay planos para la clase **CajaSimple**?
- ¿coinciden los tipos de datos asignados a los datos miembro con respecto a los especificados al declararlos?

Creación de objetos y referencias

Problemas:

- sentencias mal escritas, faltando punto y coma
- al crear un nuevo objeto se necesita llamada al constructor de la clase, que debe llamarse como ella (**Caja** y no **CajaSimple**)
- los datos miembro se declararon como enteros y sólo se permite el uso de este tipo de valores
- se indica que el método devuelve un entero, pero no hay sentencia **return**

Creación de objetos y referencias

```
// Clase para representar cajas
class Caja{
  // Datos miembro
  int alto,ancho,largo;

  // Metodos
  int calcularVolumen(){
    int resultado=alto*ancho*largo;
    return resultado;
  }
}
class TestCaja{
  // Metodo principal
  public static void main(String args[]){
    Caja caja1=new Caja();
    Caja caja2=new Caja();
    caja1.alto=23;
    caja1.ancho=2;
    caja1.largo=10;
    caja2.ancho=23;
    caja2.alto=4;

    // Se calcula el volumen de la segunda caja
    int volumen=caja2.calcularVolumen();
    System.out.println("Volumen: "+volumen);
  }
}
```

Creación de objetos y referencias

Cuestión:

¿Está todo correcto? ¿Qué resultado se escribe por pantalla?

Al ejecutar:

```
Volumen : 0
```

Error lógico

El programa compila, no hay errores de ejecución, pero el resultado parece raro. ¿Por qué?

Creación de objetos y referencias

Constructor: definición

Es el método que se llama automáticamente al crear un nuevo objeto para inicializar sus datos miembro.

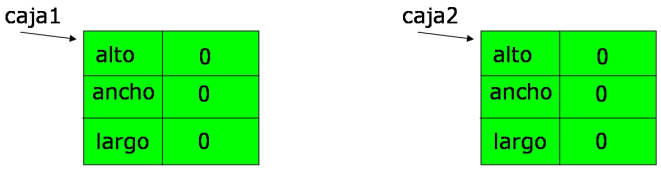
En el ejemplo visto anteriormente, no había ningún método **Caja()**. ¿Cómo puede usarse si no se ha definido?

Constructor por defecto

Si no se definen constructores de forma explícita, Java ofrece uno: el constructor por defecto, sin argumentos, que inicializa los valores de los datos miembro al valor 0 (o equivalente....)

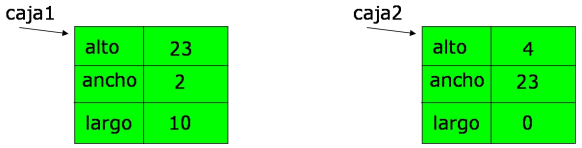
Creación de objetos y referencias

En el ejemplo de las cajas, tras la segunda llamada a **new**, tendríamos en la memoria los siguientes objetos:



Creación de objetos y referencias

Y tras la inicialización explícita de los datos miembro tendríamos:



Una vez creado el objeto, se le pueden asignar nuevos nombres (alias). Por ejemplo:

```
Caja aliasCaja2=caja2;
```

Preguntas:

- ¿Cuántos objetos hay ahora en memoria?
- ¿Es válida la siguiente sentencia?

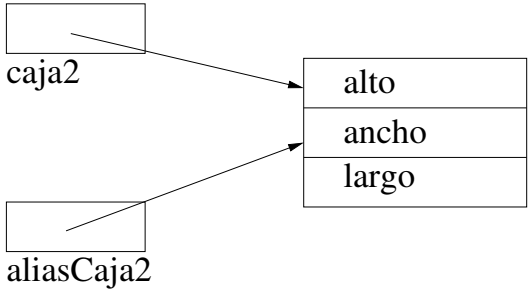
```
aliasCaja2.alto=25;
```

Creación de objetos y referencias

Referencia

Estos alias se denominan **referencias**. Las referencias NO son objetos: son etiquetas a objetos.

- Contienen una referencia hacia donde se encuentra el objeto en memoria.
- Al asignar una referencia con el valor de otra, hacemos que ambas apunten al mismo objeto.



Creación de objetos y referencias

Nota:

- El constructor por defecto sólo tiene sentido si no se define ningún constructor de forma explícita
- Pueden definirse tantos constructores como se desee
- Si se desea recuperar la existencia del constructor por defecto habrá que definirlo de forma explícita

Contenido del tema

Métodos

Definición

Acciones que pueden realizarse sobre los objetos de la clase.

- La sintaxis para definir un método es la misma de C y C++:

```
tipo nombre_de_método(lista_de_parámetros){
  // cuerpo del método
}
```

Métodos

Dentro de los métodos podemos usar los datos miembro directamente.

```
1
2  // Clase para representar cajas
3  class Caja{
4    // Datos miembro
5    int alto;
6    int ancho;
7    int largo;
8
9    // Metodos
10   int calcularVolumen(){
11     return alto*ancho*largo;
12   }
13 }
14 class TestCaja{
15   public static void main(String args[]){
16     Caja caja1=new Caja();
17     caja1.alto=2;
18     caja1.ancho=2;
19     caja1.largo=10;
20     int vol=caja1.calcularVolumen();
21
22     // Se muestra por pantalla
23     System.out.println("Volumen: "+vol);
24   }
25 }
```


Métodos con parámetros

Los métodos pueden tener parámetros al igual que en C y C++

```
class Caja{
  int alto;
  int ancho;
  int largo;

  int calcularVolumen(){
    int resultado=alto*ancho*largo;
    return resultado;
  }

  void setDim(int w, int h, int d) {
    alto = w;
    ancho= h;
    largo = d;
  }
}
class TestCaja{
  public static void main(String args[]){
    Caja caja1=new Caja();
    Caja caja2=new Caja();
    caja1.setDim(10,20,15);
    caja2.setDim(3,6,9);

    int vol=caja2.calcularVolumen();
    System.out.println("Volumen: "+vol);
  }
}
```


Operador .

Operador .

Como hemos visto en ejemplos anteriores, podemos usar una referencia y el operador . para acceder a los datos (**variables de instancia**) y métodos (**métodos de instancia**) del objeto apuntado (**objeto llamante**) por la referencia.

- `referenciaObjeto.campoDato`
- `referenciaObjeto.metodo(argumentos)`

```
class TestCaja{
  public static void main(String args[]){
    Caja caja1=new Caja();

    caja1.alto=23;
    caja1.ancho=2;
    caja1.largo=10;
    int volumen=caja2.calcularVolumen();
    System.out.println("Volumen: "+volumen);
  }
}
```


Excepción NullPointerException

NullPointerException

Se producirá una excepción `NullPointerException` si usamos una referencia con valor `null` para acceder a un dato o método miembro.

```
class Estudiante {
   String nombre; // por defecto vale null
   int edad; // por defecto vale 0
   boolean esAlumnoCiencias; //  por defecto vale false
   char genero; //  por defecto vale  '\u0000'
}
class Test {
   public static void main(String[] args) {
      Estudiante estudiante = new Estudiante();
      System.out.println("Longitud del nombre? " + estudiante.nombre.length() );
   }
}
```

Salida del programa

```
Exception in thread "main" java.lang.NullPointerException
        at Test.main(Estudiante.java:10)
```

Contenido del tema

Constructores

Constructor

Es usado para inicializar el objeto inmediatamente después de su creación.

- Tiene el mismo nombre que la clase, y no devuelve nada (ni siquiera `void`).
- Cuando no especificamos un constructor, Java crea un *constructor por defecto* , que inicializa todas las variables de instancia a cero.

Creando un constructor por defecto

```
class Caja{
  // Datos miembro
  int alto,ancho,largo;

  Caja(){
    alto=ancho=largo=10;
  }
  int calcularVolumen(){
    int resultado=alto*ancho*largo;
    return resultado;
  }
  void setDim(int w, int h, int d) {
    alto = w;
    ancho= h;
    largo = d;
  }
}
class TestCaja{
  public static void main(String args[]){
    int volumen;
    Caja caja1=new Caja();
    Caja caja2=new Caja();

    caja2.setDim(3,6,9);
    volumen=caja1.calcularVolumen();
    System.out.println("Volumen caja 1: "+volumen);
    volumen=caja2.calcularVolumen();
    System.out.println("Volumen caja 2: "+volumen);
  }
}
```


Sobrecarga de constructores

```
class Caja{
  // Datos miembro
  int alto,ancho,largo;

  Caja(){
    alto=ancho=largo=10;
  }
  Caja(int w, int h, int d) {
    alto = w;
    ancho= h;
    largo = d;
  }
  int calcularVolumen(){
    int resultado=alto*ancho*largo;
    return resultado;
  }
  void setDim(int w, int h, int d) {
    alto = w;
    ancho= h;
    largo = d;
  }
}
```

```
class TestCaja{
  public static void main(String args[]){
    int volumen;
    Caja caja1=new Caja();
    Caja caja2=new Caja(3,6,9);
    volumen=caja1.calcularVolumen();
    System.out.println("Volumen caja 1: "+volumen);
    volumen=caja2.calcularVolumen();
    System.out.println("Volumen caja 2: "+volumen);
  }
}
```

Sobrecarga de constructores

Ejercicio sobre la clase Caja

Cread un constructor para crear cajas con base cuadrada y altura cualesquiera: necesitamos dos parámetros

Sobrecarga de constructores

Un constructor puede llamar a otro constructor:

```
class Caja {
  int ancho;
  int alto;
  int largo;

  Caja(int w, int h, int d) {
    ancho = w;
    alto = h;
    largo = d;
  }
  Caja() {
    ancho = -1;//utiliza -1 para
    alto = -1; //indicar que la caja
    largo = -1;//no esta inicializada
  }
  Caja(int len) {
    this(len,len,len);
  }

  int calcularVolumen() {
    return ancho * alto * largo;
  }
}
```

```
class OverloadCons {
  public static void main(String args[]) {
    Caja caja1 = new Caja(10, 20, 15);
    Caja caja2 = new Caja();
    Caja cubo = new Caja(7);
    int vol;

    vol = caja1.calcularVolumen();
    System.out.println("El volumen de caja1 es " + vol);
    vol = caja2.calcularVolumen();
    System.out.println("El volumen de caja2 es " + vol);
    vol = cubo.calcularVolumen();
    System.out.println("El volumen de cubo es " + vol);
  }
}
```


Contenido del tema

La referencia this

La referencia this

Los métodos de una clase pueden referenciar al objeto que lo invocó con la palabra clave **this**.

```
1  class Caja{
2    // Datos miembro
3    int alto,ancho,largo;
4
5    Caja(int ancho,int alto,int largo) {
6      this.ancho = ancho;
7      this.alto = alto;
8      this.largo = largo;
9    }
10
11 }
```

Contenido del tema

Ejemplo de clase: clase Pila I

- La clase permite el encapsulamiento de datos y código.
- La clase es como una *caja negra*: No hace falta saber qué ocurre dentro para poder utilizarla.

```
1  class Pila {
2    int pila[] = new int[10];
3    int tope;
4    Pila() {/*Inicializa la posicion superior de la pila*/
5      tope = -1;
6    }
7    void push(int item) {/*Introduce un elemento en la pila*/
8      if(tope==9)
9        System.out.println("La pila esta llena");
10     else
11       pila[++tope] = item;
12   }
13   int pop() {/*Extrae un elemento de la pila*/
14     if(tope < 0) {
15       System.out.println("La pila esta vacia");
16       return 0;
17     }
18     else
19       return pila[tope--];
20   }
21 }
22
```


Ejemplo de clase: clase Pila II

```
23
24 class TestPila {
25   public static void main(String args[]) {
26     Pila mipila1 = new Pila();
27     Pila mipila2 = new Pila();
28     // introduce algunos numeros en la pila
29     for(int i=0; i<10; i++) mipila1.push(i);
30     for(int i=10; i<20; i++) mipila2.push(i);
31     // extrae los numeros de la pila
32     System.out.println("Contenido de la pila mipila1:");
33     for(int i=0; i<10; i++)
34        System.out.println(mipila1.pop());
35     System.out.println("Contenido de la pila mipila2:");
36     for(int i=0; i<10; i++)
37        System.out.println(mipila2.pop());
38   }
39 }
```


Contenido del tema

Sobrecarga de métodos

Sobrecarga de métodos

Consiste en que dos o más métodos de una clase tienen el mismo nombre, pero con listas de parámetros distintos.

- La sobrecarga es usada para implementar el *polimorfismo*.
- Java utiliza el tipo y/o el número de argumentos como guía para ver a cual método llamar.
- El tipo que devuelve un método es insuficiente para distinguir dos versiones de un método.

Sobrecarga de métodos I

Ejemplo de sobrecarga

```
1  class OverloadDemo {
2    void test() {
3      System.out.println("Sin parametros");
4    }
5
6    // Sobrecarga el metodo test con un parametro entero
7   void test(int a) {
8      System.out.println("a: " + a);
9    }
10
11   // Sobrecarga el metodo test con dos parametros enteros
12   void test(int a, int b) {
13     System.out.println("a y b: " + a + " " + b);
14   }
15
16   // Sobrecarga el metodo test con un parametro double
17   double test(double a) {
18     System.out.println("double a: " + a);
19     return a*a;
20   }
21 }
22
23
```


Sobrecarga de métodos II

```
24 class Overload {
25   public static void main(String args[]) {
26     OverloadDemo ob = new OverloadDemo();
27     double result;
28
29     // llama a todas las versiones de test()
30     ob.test();
31     ob.test(10);
32     ob.test(10, 20);
33     result = ob.test(123.2);
34     System.out.println("Resultado de ob.test(123.2): " + result);
35   }
36 }
```

```
Sin parametros
a: 10
a y b: 10 20
double a: 123.2
Resultado de ob.test(123.2): 15178.240000000002
```


Sobrecarga con conversión automática de tipos

- Java busca una versión del método cuyos parámetros actuales coincidan con los parámetros formales, en número y tipo.
- Si el tipo no es exacto puede que se aplique *conversión automática de tipos*.

Ejemplo

```
1  class OverloadDemo {
2    void test() {
3      System.out.println("Sin parametros");
4    }
5
6    // Sobrecarga el metodo test con dos parametros enteros
7   void test(int a, int b) {
8      System.out.println("a y b: " + a + " " + b);
9    }
10
11   // Sobrecarga el metodo test con un parámetro double
12   void test(double a) {
13     System.out.println("Dentro de test(double) a: " + a);
14   }
15 }
16
17 class Overload {
18   public static void main(String args[]) {
19     OverloadDemo ob = new OverloadDemo();
20     int i = 88;
21
22     ob.test();
23     ob.test(10, 20);
24
25     ob.test(i); // esto llama a test(double)
26     ob.test(123.2); // esto llama a test(double)
27   }
28 }
```


```
Sin parametros
a y b: 10 20
Dentro de test(double) a: 88.0
Dentro de test(double) a: 123.2
```


Contenido del tema

Objetos como parámetros

Un objeto de una determinada clase puede ser parámetro de un método de esa u otra clase.

```
1  class Test {
2    int a, b;
3
4    Test(int i, int j) {
5      a = i;
6      b = j;
7    }
8
9    // devuelve true si o es igual al objeto llamante
10   boolean equals(Test o) {
11     if(o.a == a && o.b == b) return true;
12     else return false;
13   }
14 }
15
16 class PassOb {
17   public static void main(String args[]) {
18     Test ob1 = new Test(100, 22);
19     Test ob2 = new Test(100, 22);
20     Test ob3 = new Test(-1, -1);
21     System.out.println("ob1 == ob2: " + ob1.equals(ob2));
22     System.out.println("ob1 == ob3: " + ob1.equals(ob3));
23   }
24 }
```

Objetos como parámetros

Otro ejemplo: Constructor de copia

Suele usarse para hacer una copia de otro objeto.

```
1 Caja(Caja ob) {
2   ancho = ob.ancho;
3   alto = ob.alto;
4   largo = ob.largo;
5 }
6
7 Caja caja = new Caja(10,20,15);
8 Caja miclone = new Caja(caja);
```


Contenido del tema

Paso por valor y referencia

Cuando se pasan datos a un método, como argumentos, éstos pueden ser:

- Datos de tipos de datos primitivo
- Objetos de alguna clase

Paso por valor: Argumento de tipo primitivo

En este caso se habla de **paso por valor**. En realidad, el método realiza una copia propia del valor pasado como argumento. Esto implica que:

- Una vez hecha la copia no hay más relación entre el valor pasado y el usado en el método, que será una copia independiente del primero
- Por esta razón, si en el interior del método se produce algún cambio en el valor del argumento, no tiene efecto sobre el valor pasado

Paso por valor y referencia: valor de tipo primitivo I

```
1  // Clase de ejemplo de paso por valor
2  class PasoValor{
3    // Metodo que eleva al cuadrado un valor
4    int elevarCuadrado(int valor){
5      // Se eleva al cuadrado el argumento
6      valor=valor*valor;
7
8      // Se devuelve su valor
9      return valor;
10   }
11
12   // Metodo main
13   public static void main(String args[]){
14     // Se crea objeto de la clase, para poder hacer uso del metodo
15     PasoValor objeto=new PasoValor();
16
17     // La variable a contiene el valor a elevar
18     int a=25;
19
20     // Se usa el metodo
21     int resultado=objeto.elevarCuadrado(a);
22
23     // Al final a sigue valiendo 25, ya que el metodo elevarCuadrado
24     // usa copia local (valor) y no realiza ningún cambio sobre a
25     System.out.println(a+" elevado al cuadrado es: "+resultado);
26   }
27 }
```

Paso por valor y referencia: objeto de una clase

Paso por referencia: Argumento objeto de una clase

En este caso se habla de **paso por referencia** (lo que se pasa al método es en realidad un alias, es decir, una forma de acceder al objeto). Esto implica que:

- el objeto apuntado por la referencia NO se copia: está él mismo accesible desde el método gracias a la referencia
- por esta razón, puede ser modificado haciendo uso de sus datos y métodos
- los cambios realizados en el objeto serán permanentes (no desaparecen al finalizar la ejecución del método)

Paso por valor y referencia: objeto de una clase

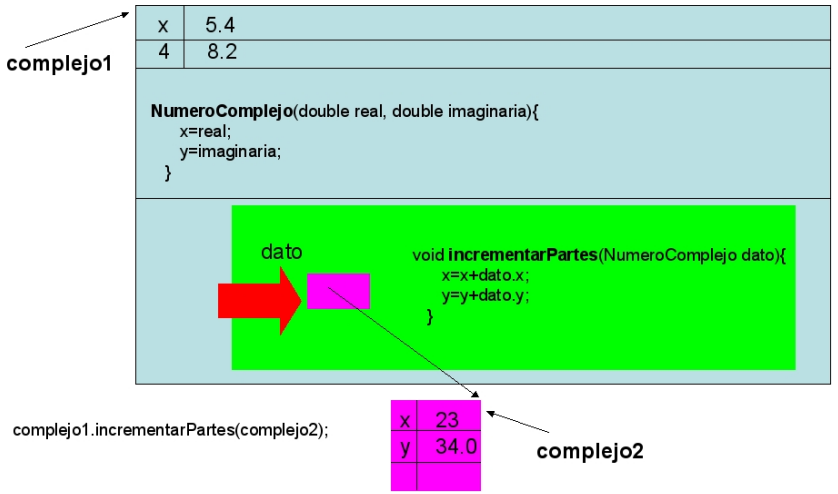
```
// Clase para representar numeros complejos: se usa tambien
// para ilustrar el paso por referencia
class NumeroComplejo{
  // Datos miembros
  private double x; // Parte real
  private double y; // Parte imaginaria

  // Constructor de la clase
  NumeroComplejo(double real, double imaginaria){
    x=real;
    y=imaginaria;
  }

  // Metodo que incrementa el valor del objeto con los
  // valores del objeto pasado como argumento
  void incrementarPartes(NumeroComplejo dato){
    x=x+dato.x;
    y=y+dato.y;
  }

  // Metodo para imprimir valores de datos miembros
  public void imprimir(){
    System.out.println(" --- Objeto clase NumeroComplejo ---");
    System.out.println("x: "+x+" y: "+y);
    System.out.println(".....................................");
  }
}
```


Paso por valor y referencia: objeto de una clase

```
class TestNumeroComplejo{
  public static void main(String args[]){
    NumeroComplejo complejo1=new NumeroComplejo(5.4,8.2);
    NumeroComplejo complejo2=new NumeroComplejo(23,34.0);

    // Se incrementa el primero, pasando como argumento el
    // segundo
    complejo1.incrementarPartes(complejo2);

    // Se muestra el primero, para ver si se incremento
    complejo1.imprimir();
  }
}
```


Paso por valor y referencia: objeto de una clase)

La situación podría representarse de la siguiente forma:



Paso por valor y referencia: objeto de una clase

En este caso el efecto del método ha sido modificar el valor de los datos miembro del objeto sobre el que se produce la llamada. Es decir, el método no devuelve nada (**void**): todo lo que hay que modificar está al alcance de Java en el momento de ejecutar.

Aquí no se produce cambio alguno sobre el valor del objeto pasado como argumento, pero ¿qué ocurre si dentro del método hiciéramos lo siguiente?

```
void incrementarPartes(NumeroComplejo dato){
    x=x+dato.x;
    y=y+dato.y;
    dato.x=0;
    dato.y=0;
}
```


Paso por valor y referencia: objeto de una clase

¿Y si se cambia dentro del método el lugar al que apunta la referencia dato?

```
void incrementarPartes(NumeroComplejo dato){
    x=x+dato.x;
    y=y+dato.y;
    NumeroComplejo nuevo=new NumeroComplejo(5,5);
    dato=nuevo;
}
```


Paso por valor y referencia: objeto de una clase

```
// Clase para representar numeros complejos: se usa tambien
// para ilustrar el paso por referencia
class NumeroComplejo{
  // Datos miembros
  private double x; // Parte real
  private double y; // Parte imaginaria

  // Constructor de la clase
  NumeroComplejo(double real, double imaginaria){
    x=real;
    y=imaginaria;
  }

  // Metodo que incrementa el valor del objeto con los
  // valores del objeto pasado como argumento
  void incrementarPartes(NumeroComplejo dato){
    x=x+dato.x;
    y=y+dato.y;
    NumeroComplejo nuevo=new NumeroComplejo(5,5);
    dato=nuevo;
  }

  // Metodo para imprimir valores de datos miembros
  public void imprimir(){
    System.out.println(" --- Objeto clase NumeroComplejo ---");
    System.out.println("x: "+x+" y: "+y);
    System.out.println(".....................................");
  }
}
```


Paso por valor y referencia: objeto de una clase

```
class TestNumeroComplejo{
  public static void main(String args[]){
    NumeroComplejo complejo1=new NumeroComplejo(5.4,8.2);
    NumeroComplejo complejo2=new NumeroComplejo(23,34.0);

    // Se incrementa el primero, pasando como argumento el segundo
    complejo1.incrementarPartes(complejo2);

    // Se muestra el primero, para ver si se incremento
    complejo1.imprimir();

    // Se muestra el segundo, para ver si cambio
    complejo2.imprimir();
  }
}
```

Paso por valor y referencia: objeto de una clase

```
 --- Objeto clase NumeroComplejo ---
x: 28.4 y: 42.2
.....................................
 --- Objeto clase NumeroComplejo ---
x: 23.0 y: 34.0
.....................................
```


Paso por valor y referencia: objeto de una clase

Nota:

Como se ve, no se produce cambio en el valor de los datos miembros del objeto pasado como argumento. Esto indica que no puede modificarse la referencia: no podemos cambiar a donde apunta.

Esto no coincide con lo que ocurre con otros lenguajes de programación: el paso por referencia permite hacer este tipo de cosas.

Contenido del tema

Control de acceso

Definición:

Posibilidad de controlar el grado de visibilidad de los datos y de los métodos

Sentido:

Preservar información interna a la clase evitando que pueda accederse o usarse desde fuera.

Control de acceso

Los especificadores de acceso para datos miembro y métodos son:

- **private**: Sólo es accesible por miembros de la misma clase.
- **public**: Accesible por miembros de cualquier clase.
- **protected**: Está relacionado con la herencia.
- **Por defecto**: si no se indica nada los miembros son públicos dentro de su mismo paquete

Ejemplo

```
1  class Test {
2    int a; // acceso por defecto
3    public int b; // acceso publico
4    private int c; // acceso privado
5
6    void setc(int i) { // establece el valor de c
7      c = i;
8    }
9    int getc() { // obtiene el valor de c
10     return c;
11   }
12 }
13 class AccessTest {
14   public static void main(String args[]) {
15     Test ob = new Test();
16
17     // Esto es correcto, a y b pueden ser accedidas directamente
18     ob.a = 10;
19     ob.b = 20;
20
21     // Esto no es correcto y generara un error de compilacion
22 //   ob.c = 100; // Error!
23
24     // Se debe acceder a c a traves de sus metodos
25     ob.setc(100); // OK
26     System.out.println("a, b, y c: " + ob.a + " " +
27                        ob.b + " " + ob.getc());
28   }
29 }
```


Contenido del tema

Ocultamiento de datos miembro

Ocultamiento de datos miembro

Es conveniente hacer que los datos miembro de una clase sean privados:

- Para protegerlos frente a **modificaciones incorrectas**.
- Para hacer que la clase sea **fácil de mantener**.

Ocultamiento de datos miembro

Ejemplo de objeto en estado incorrecto: clase Fecha

La siguiente clase `Fecha` tiene los datos públicos, con lo que pueden ser modificados desde otra clase, permitiendo dejarlos en un **estado incorrecto**.

```
class Fecha{
   public int dia;
   public int mes;
   public int anio;

   Fecha(){
     dia=1;
     mes=1;
     anio=2000;
   }
}
```

```
class TestFecha{
   public static void main(String[] args){
       fecha = new Fecha();
       fecha.dia=30; //fecha incorrecta
       fecha.mes=2;
   }
}
```


Ocultamiento de datos miembro

Solución: clase Fecha

Hacemos los datos privados e incluimos métodos de consulta (`get`) y métodos de modificación (`set`) que comprueben que los cambios mantienen el objeto en un **estado consistente**.

```
class Fecha{
   private int dia;
   private int mes;
   private int anio;

   Fecha(){
     dia=1;
     mes=1;
     anio=2000;
   }
   void setFecha(int nuevoDia, int nuevoMes,
                 int nuevoAnio){
      // Si la nueva fecha es correcta
         // la cambiamos
   }
   int getDia()  { return dia; }
   int getMes() { return mes; }
   int getAnio() { return anio; }
}
```

```
// Programa que usa clase Fecha
class TestFecha{
   public static void main(
                String[] args){
      fecha = new Fecha();
      fecha.setFecha(30,2,2000);
   }
}
```


Ocultamiento de datos miembro

Ejemplo de clase difícil de mantener: clase Punto

La siguiente clase `Punto` tiene los datos públicos, con lo que es **difícil de mantener**: un cambio en los datos miembro hace que tengan que modificarse todas las clases que la usan.

```
public class Punto{
// Punto representado
// con coord. cartesianas
  public double x;
  public double y;
}
```

```
// Programa que usa la clase Punto
class TestPunto{
   public static void main(
                String[] args){
      punto = new Punto();
      punto.x=3.0;
      punto.y=4.0;
   }
}
```


Ocultamiento de datos miembro

Ejemplo de clase difícil de mantener: clase Punto

Es posible que algún día decidamos modificar los datos miembro que representan un Punto: obligaría a cambiar todas las demás clases que usan Punto. Esto es muy **sensible a errores**.

```
public class Punto{
// Punto representado
// con coord. polares
  public double r;
  public double alfa;
}
```

```
// Programa que usa la clase Punto
class TestPunto{
   public static void main(
                String[] args){
      punto = new Punto();
      punto.r=5.0;
      punto.alfa=0.9273;
   }
}
```

Ocultamiento de datos miembro

Solución: clase Punto

Usamos **datos privados** y un **interfaz bien definido** mediante métodos de consulta (`get`) y modificación (`set`): un cambio de representación solo necesita cambiar los métodos de la clase Punto. Las clases que usan Punto **no tienen que cambiarse**.

```
public class Punto{
// Punto representado
// con coord. cartesianas
  private double r; // modulo
  private double alfa; // angulo
  public double getX(){
     return r * Math.cos(alfa);
  }
  public double getY(){
     return r * Math.sin(alfa);
  }
  public double getModulo(){
     return r;
  }
  public double getAngulo(){
     return alfa;
  }
```

```
  void setXY(double x, double y){
     r=Math.sqrt(x*x+y*y);
     alfa=Math.atan(y/x);
  }
  void setModuloAngulo(double modulo, double angulo){
     r=modulo;
     alfa=angulo;
  }
}
```

```
// Programa que usa la clase Punto
class TestPunto{
   public static void main(String[] args){
      punto = new Punto();
      punto.setXY(3.0,4.0);
   }
}
```


Contenido del tema

Modificador static

Al hablar del método **main** ya se ha comentado el efecto de agregar la palabra reservada **static**. En realidad esta palabra puede usarse sobre cualquier método y sobre cualquier dato miembro.

Efecto:

El efecto de agregar la palabra reservada **static** a un método o a un dato miembro es hacer que dicho elemento esté asociado a la clase y no a los objetos individuales.
Un miembro **static** puede ser usado sin crear ningún objeto de su clase

Métodos static

Sentido de los métodos estáticos:

Métodos cuyo trabajo no está centrado en objetos, al ser generales. Por ejemplo, en la clase **Math**, encargada de ofrecer cálculo de funciones matemáticas, los métodos se declaran como estáticos. El cálculo del valor absoluto, o del seno, no están directamente relacionados con objetos individuales.

Restricciones de los métodos estáticos:

- Sólo pueden llamar a métodos **static**
- Sólo deben acceder a datos **static**
- No pueden usar **this** o **super**

Métodos static

```
1
2  // Clase para mostrar la forma de usar métodos estáticos
3  class EjemploAbs{
4    public static void main(String args[]){
5      int a=-34;
6      double b=-4356.78;
7      long c=93456;
8
9      // Se calculan los valores absolutos
10     int vabsa=Math.abs(a);
11     System.out.println("|a| = "+vabsa);
12     double vabsb=Math.abs(b);
13     System.out.println("|b| = "+vabsb);
14     long vabsc=Math.abs(c);
15     System.out.println("|c| = "+vabsc);
16   }
17 }
```

Métodos static

Observad que:

- No hemos creado objeto alguno
- El método **abs** se llama sobre la clase: **Math.abs()**
- Al ser general, se puede usar con cualquier tipo de dato primitivo (si no fuera así, sería necesario incluir el método en cada clase particular: la asociada a cada tipo de dato para el que se desea disponer de esta información)

Datos static

Datos static

Variables asociadas a la clase completa y no a objetos particulares. Existe una sola copia de la variable para todos los objetos de su clase.

Imaginemos que se desea disponer del día y mes de la creación de una clase. Estas características son propias de la clase en sí y no de los objetos individuales. Por esta razón, se declaran como estáticos.

Datos static

```
1  // Clase con datos miembros estaticos
2  class DatosEstaticos{
3    // Dato miembro para indicar el dia de creacion de la clase
4    static int diaCreacionClase=21;
5
6    // Dato miembro para indicar el mes de creacion de la clase
7    static int mesCreacionClase=11;
8
9    // Dato miembro particular para cada objeto
10   int identificadorObjeto;
11
12   // Constructor
13   DatosEstaticos(int identificador){
14     identificadorObjeto=identificador;
15   }
16
17   // Metodo main
18   public static void main(String args[]){
19     DatosEstaticos objeto1=new DatosEstaticos(1);
20     DatosEstaticos objeto2=new DatosEstaticos(2);
21     DatosEstaticos objeto3=new DatosEstaticos(3);
22   }
23 }
```

Datos static

Si pudiera ver la memoria del ordenador tras la creación del tercer objeto tendría algo parecido a lo siguiente:

Datos static

Importante:

- el espacio de almacenamiento para **diaCreacionClase** y **mesCreacionClase** no depende de los objetos, sino de la misma clase
- todos los objetos de la clase comparten estos valores
- si se modifican estos datos, quedan modificados de forma global
- la inicialización de los datos debe hacerse al declararlos, y no en el constructor (si fuera así se cambiaría su valor al crear cada objeto)

Modificador static

Formas de aludir a miembros estáticos:

- como si se tratara de miembros normales: **objeto.dato**, **objeto.metodo()**
- eliminar la referencia al objeto (indicando que es algo de la clase): **dato**, **metodo()**
- aludiendo a ellos a través del nombre de la clase: **Clase.dato**, **Clase.metodo()**

La segunda forma sólo puede usarse cuando estamos en la propia clase en que se declararon los miembros estáticos. La última forma se ha usado al calcular el valor absoluto (método declarado en la clase **Math**) desde el código de la clase **EjemploAbs**.

Modificador static

```
1  // Clase para mostrar la forma de uso de los miembros estaticos
2  // cuando se usan desde la clase en que se declaran
3  class UsoEstatico{
4    static int diaCreacionClase=21;
5    static int mesCreacionClase=11;
6
7    int identificadorObjeto;
8
9    // Constructor de la clase
10   UsoEstatico(int identificador){
11     identificadorObjeto=identificador;
12   }
13
14   public static void main(String args[]){
15     UsoEstatico objeto1=new UsoEstatico(1);
16     UsoEstatico objeto2=new UsoEstatico(2);
17     UsoEstatico objeto3=new UsoEstatico(3);
18
19     // Uso habitual: objeto.dato
20     System.out.println("diaCreacionClase: "+objeto1.diaCreacionClase);
21
22     // Uso sin objeto
23     System.out.println("mesCreacionClase: "+mesCreacionClase);
24
25     // Uso con nombre de clase
26     System.out.println("mesCreacionClase: "+UsoEstatico.mesCreacionClase);
27   }
28 }
```

Modificador static

```java
// Clase para demostrar el uso de datos miembros estaticos
// desde fuera de la clase en que se declaran
class UsoEstaticoExterno{
  public static void main(String args[]){
    // Se puede acceder a los datos miembros estaticos
    // incluso sin haber creado objetos
    System.out.println("Dia creacion clase: "+UsoEstatico.diaCreacionClase);
    System.out.println("Mes creacion clase: "+UsoEstatico.mesCreacionClase);

    // Tambien puede accederse a traves de objeto
    UsoEstatico objeto=new UsoEstatico(4);
    System.out.println("Mes creacion: "+objeto.mesCreacionClase);
  }
}
```

Bloques de código static

Bloque de código static

Una clase puede tener un bloque static que se ejecuta una sola vez cuando la clase se carga por primera vez.
Puede usarse por ejemplo para inicializar los datos static.

Modificador static

```java
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
  static {
    System.out.println("Bloque estático inicializado.");
    b = a * 4;
  }
  public static void main(String args[]) {
    meth(42);
  }
}
```

```
Bloque estático inicializado.
x = 42
a = 3
b = 12
```


Contenido del tema

Especificador final con datos

Variable **final**

Una variable **final** es una variable a la que no podemos modificar su contenido.

- Puede ser una constante definida en tiempo de compilación, o bien, en tiempo de ejecución que no se cambiará.
- Son similares al **const** de C/C++
- Suele utilizar identificadores en mayúscula.
- Las variables final suelen ser también static (existe una sola copia de ellas para todos los objetos de la clase).
- **final** se puede aplicar a:
  - Datos primitivos: Significa que el valor no se cambiará.
  - Referencias a objetos: Significa que la referencia no se cambiará, aunque sí podemos cambiar el contenido del objeto.
  - Parámetros de un método.

Especificador final I

Ejemplo de final con datos primitivos y referencias

```
1  class Value {
2    int i = 1;
3  }
4  public class FinalData {
5    // constantes inicializadas en tiempo de compilación
6    final int i1 = 9;
7    static final int VAL_TWO = 99;
8    // Uso típico de constante pública:
9    public static final int VAL_THREE = 39;
10   // constantes inicializadas en tiempo de ejecución
11   final int i4 = (int)(Math.random()*20);
12   static final int i5 = (int)(Math.random()*20);
13
14   Value v1 = new Value();
15   final Value v2 = new Value();
16   static final Value v3 = new Value();
17   // Arrays:
18   final int[] a = { 1, 2, 3, 4, 5, 6 };
19
20   public void print(String id) {
21     System.out.println(
22       id + ": " + "i4 = " + i4 + ", i5 = " + i5);
23   }
```


Especificador final II

```
24   public static void main(String[] args) {
25     FinalData fd1 = new FinalData();
26     //! fd1.i1++; // Error: no se puede cambiar valor
27     fd1.v2.i++; // Objecto no es constante!
28     fd1.v1 = new Value(); // OK -- no es final
29     for(int i = 0; i < fd1.a.length; i++)
30       fd1.a[i]++; // Objecto no es constante!
31     //! fd1.v2 = new Value(); // Error: No se puede
32     //! fd1.v3 = new Value(); // cambiar referencia
33     //! fd1.a = new int[3];
34
35     fd1.print("fd1");
36     System.out.println("Creando nuevo objeto FinalData");
37     FinalData fd2 = new FinalData();
38     fd1.print("fd1");
39     fd2.print("fd2");
40   }
41 } ///:~
```

```
fd1: i4 = 7, i5 = 0
Creando nuevo objeto FinalData
fd1: i4 = 7, i5 = 0
fd2: i4 = 3, i5 = 0
```


Especificador final

Ejemplo de final con parámetros de un método

```
1  class Gizmo {
2    public void spin() {}
3  }
4
5  public class FinalArguments {
6    void with(final Gizmo g) {
7      //! g = new Gizmo(); // Ilegal -- g es final
8      g.spin();
9    }
10   void without(Gizmo g) {
11     g = new Gizmo(); // OK -- g no es final
12     g.spin();
13   }
14   // void f(final int i) { i++; } // No se puede cambiar
15   // Sólo se puede leer un dato primitivo que sea final
16   int g(final int i) { return i + 1; }
17   public static void main(String[] args) {
18     FinalArguments bf = new FinalArguments();
19     Gizmo g = new Gizmo();
20     bf.without(g);
21     bf.with(g);
22   }
23 } ///:~
```

Constantes blancas I

Constante blanca

Dato declarado como **final** pero al que no se da un valor de inicialización en la declaración. Tendrá que inicializarse obligatoriamente en el constructor.

Constantes blancas II

```
1  //: c06:BlankFinal.java
2  // From 'Thinking in Java, 2nd ed.' by Bruce Eckel
3  // www.BruceEckel.com. See copyright notice in CopyRight.txt.
4  // "Blank" final data members.
5
6  class Poppet { }
7
8  class BlankFinal {
9    final int i = 0; // Initialized final
10   final int j; // Blank final
11   final Poppet p; // Blank final reference
12   // Blank finals MUST be initialized
13   // in the constructor:
14   BlankFinal() {
15     j = 1; // Initialize blank final
16     p = new Poppet();
17   }
18   BlankFinal(int x) {
19     j = x; // Initialize blank final
20     p = new Poppet();
21   }
22   public static void main(String[] args) {
23     BlankFinal bf = new BlankFinal();
24   }
25 } ///:~
```


Contenido del tema

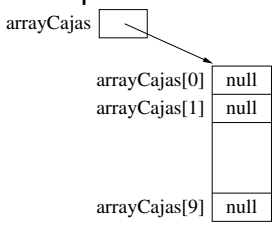
Arrays de objetos

Arrays de objetos

Puede crearse un array de objetos de cualquier clase, al igual que un array de un tipo primitivo.

```
Caja[] arrayCajas = new Caja[10];
```

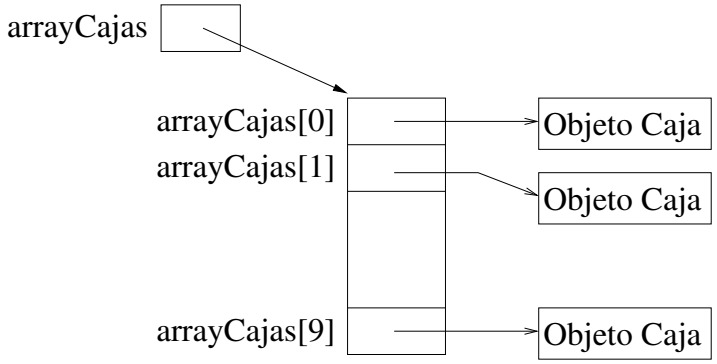
La anterior sentencia solo crea el array donde cada elemento es una variable referencia con el valor por defecto `null`.



Arrays de objetos

Los objetos de un array deben crearse explícitamente en un paso posterior:

```
for(int i=0; i<arrayCajas.lenght; i++){
   arrayCajas[i] = new Caja();
}
```



Contenido del tema

Clase String

Es una clase muy usada, que sirve para almacenar cadenas de caracteres (incluso los literales).

- Los objetos **String** no pueden modificar su contenido.
- Los objetos **StringBuffer** sí que pueden modificarse.
- El operador **+** permite concatenar cadenas.
- El método **boolean equals(String objeto)** compara si dos cadenas son iguales.
- El método **int length()** obtiene la longitud de una cadena.
- El método **char charAt(int pos)** obtiene el carácter que hay en la posición **pos**.

Clase String

```
1  // Muestra la utilizacion de algunos metodo de la clase String
2  class StringDemo2 {
3    public static void main(String args[]) {
4      String strOb1 = "Primera cadena";
5      String strOb2 = "Segunda cadena";
6      String strOb3 = strOb1;
7      System.out.println("La longitud de strOb1 es: " +
8                         strOb1.length());
9      System.out.println("El caracter de la posicion 3 de strOb1 es: " +
10                        strOb1.charAt(3));
11     if(strOb1.equals(strOb2))
12       System.out.println("strOb1 == strOb2");
13     else
14       System.out.println("strOb1 != strOb2");
15     if(strOb1.equals(strOb3))
16       System.out.println("strOb1 == strOb3");
17     else
18       System.out.println("strOb1 != strOb3");
19   }
20 }
```

Contenido del tema

Argumentos de la línea de órdenes

Al ejecutar un programa Java, podemos pasarle una lista de argumentos por la línea de comandos (al igual que en C y C++)

```
1 // Presenta todos los argumentos de la linea de ordenes
2 class CommandLine {
3   public static void main(String args[]) {
4     for(int i=0; i<args.length; i++)
5       System.out.println("args[" + i + "]: " +
6                          args[i]);
7   }
8 }
```

Si ejecutamos este programa con:

```
java CommandLine esto es una prueba 100 -1
```

obtendremos como salida:

```
args[0]: esto
args[1]: es
args[2]: una
args[3]: prueba
args[4]: 100
args[5]: -1
```


Contenido del tema

Abstracción de clases y encapsulamiento

Abstracción de clases

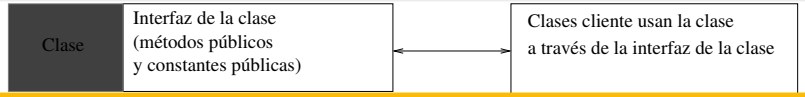
Significa separar la implementación de una clase de la forma en que se usa. Los detalles de la implementación se encapsulan y ocultan al usuario.

- La colección de métodos y campos de una clase que son accesibles desde fuera de ella (junto con una descripción de cómo estos datos y métodos miembro deben comportarse) define el ***contrato de la clase*** (también llamado ***interfaz de la clase***).
- Los detalles de la implementación se encapsulan y ocultan al usuario de la clase: ***encapsulamiento de la clase***.
- De esta forma, podremos usar una clase sin conocer los detalles de cómo está implementada.
- Una clase es por ello también conocida como un ***tipo de dato abstracto***.

Abstracción de clases y encapsulamiento

Clase como caja negra

Debemos pensar en una clase como una caja negra que oculta a las clases cliente la forma en que funciona internamente.



Analogía con un ejemplo del mundo real

La construcción de un ordenador necesita de muchos componentes (CPU, memoria, disco duro, placa madre, ventilador, etc).

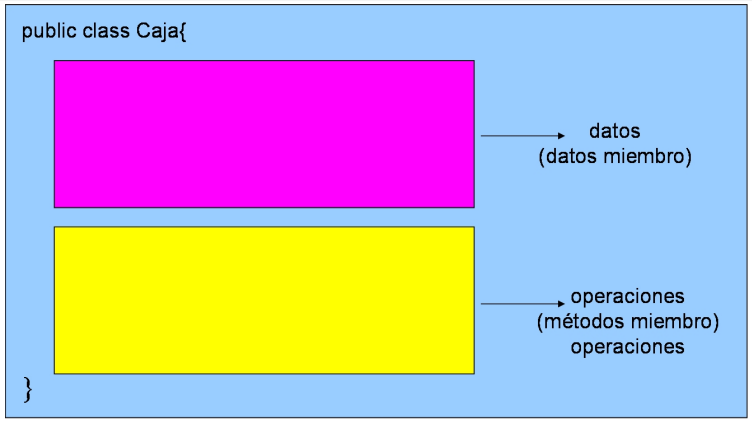
- Para montar estos componentes, no necesitamos saber cómo funciona internamente cada uno. Solo necesitamos saber cómo se usa cada uno y cómo conectarlo a los demás.
- La implementación interna es encapsulada y ocultada.
- Podemos construir un ordenador sin saber cómo está implementado cada componente.

Contenido del tema

Pensando en objetos

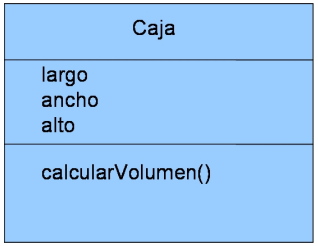
Programación dirigida a objetos

Permite agrupar datos y métodos como un todo. Ni los datos ni los métodos existen de forma independiente.



Pensando en objetos

La representación gráfica usada (UML) pone de relieve este hecho.



Pensando en objetos

La programación orientada a objetos es opuesta al paradigma procedimental, donde no hay unión clara entre datos y métodos (no hay encapsulamiento).
En el **paradigma procedimental**:

- Se centra en el diseño de métodos (funciones).
- No hay representación de elementos concretos (no hay objetos)
- Los datos, por tanto, existen de forma aislada
- Los métodos existen de forma aislada
- Los datos se hacen llegar a los métodos para que trabajen: ésta es toda la relación existente

Pensando en objetos

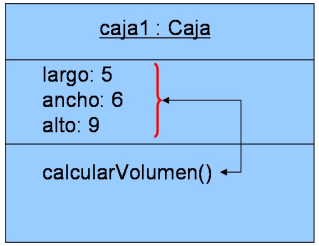
En **orientación a objetos**:

- Se unen datos y métodos en los objetos.
- Hay representación de los objetos (cajas, por ejemplo). Cada objeto representa una caja en concreto
- Los datos pertenecen (caracterizan) a un objeto
- Los métodos pertenecen (operan) sobre un objeto
- Los métodos trabajan sobre un objeto: los datos usados serán los del objeto

Pensando en objetos

Si disponemos de un objeto llamado caja1, la forma de calcular el volumen sería:

```
caja1.calcularVolumen();
```



Pensando en objetos

Resumen:

- Sin orientación a objetos el mismo método se usa para calcular un volumen con unos datos cualesquiera (sin asociación clara...)

```
calcularVolumen(3,4,8);
.......................
calcularVolumen(3.4,8.9,2.5);
```

- Se trata de una relación de uso
- ¿Qué es más claro?

```
P. Proc.: dispara(pepe,luis);
.......................
O. Objetos: pepe.dispara(luis);
```

Pensando en objetos

Cálculo del índice de masa corporal

Usando solo el **paradigma procedimental** construiríamos:

```
import java.util.Scanner;
public class CalcularEInterpretarBMI2 {
  public static double getBMI(double pesoEnKgs, double alturaEnMetros){
     return pesoEnKgs / (alturaEnMetros * alturaEnMetros);
  }
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Introduce peso en kilogramos: ");
    double peso = input.nextDouble();
    System.out.print("Introduce altura en metros: ");
    double altura = input.nextDouble();
    double bmi = getBMI(peso, altura);
    System.out.println("BMI es " + bmi);
    if (bmi < 18.5)
      System.out.println("Bajo peso");
    else if (bmi < 25)
      System.out.println("Normal");
    else if (bmi < 30)
      System.out.println("Sobrepeso");
    else
      System.out.println("Obeso");
  }
}
```


Pensando en objetos

- El anterior código tiene la ventaja de que el método `getBMI` lo podemos reutilizar en otros programas.
- Sin embargo tiene limitaciones: los datos (peso y altura) de una persona se dan de **forma aislada**.
- Supongamos que queremos también asociar un nombre y edad a una persona. Lo mejor sería agrupar todos los datos sobre una persona en un objeto.

Pensando en objetos I

Cálculo del índice de masa corporal

Usando el **paradigma orientado a objetos** construiríamos:

```
public class BMI {
  private String nombre;
  private int edad;
  private double peso; // en kilogramos
  private double altura; // en metros

  public BMI(String nombre, int edad, double peso, double altura) {
    this.nombre = nombre;
    this.edad = edad;
    this.peso = peso;
    this.altura = altura;
  }

  public BMI(String nombre, double peso, double altura) {
    this(nombre, 20, peso, altura);
  }

  public double getBMI() {
    double bmi = peso / (altura * altura);
    return Math.round(bmi * 100) / 100.0;
  }
```


Pensando en objetos II

```
  public String getEstado() {
    double bmi = getBMI();
    if (bmi < 18.5)
      return "Bajo peso";
    else if (bmi < 25)
      return "Normal";
    else if (bmi < 30)
      return "Sobrepeso";
    else
      return "Obeso";
  }

  public String getNombre() {
    return nombre;
  }

  public int getEdad() {
    return edad;
  }

  public double getPeso() {
    return peso;
  }

  public double getAltura() {
    return altura;
  }
}
```

Pensando en objetos

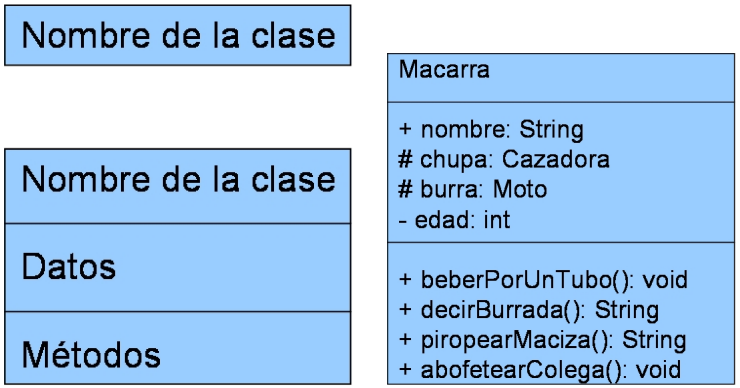
```
public class UsarClaseBMI {
 public static void main(String[] args) {
   BMI bmi1 = new BMI("Juan Zambrano", 18, 63, 1.79);
   System.out.println("El BMI para " + bmi1.getNombre() + " es "
     + bmi1.getBMI() + " " + bmi1.getEstado());

   BMI bmi2 = new BMI("Alberto Olmedo", 105, 1.65);
   System.out.println("El BMI para " + bmi2.getNombre() + " es "
     + bmi2.getBMI() + " " + bmi2.getEstado());
 }
}
```

```
El BMI para Juan Zambrano es 19.66 Normal
El BMI para Alberto Olmedo es 38.57 Obeso
```


Contenido del tema

UML: Unified Modeling Language

Estudiaremos sólo cómo se representan clases y objetos y las relaciones más usuales entre clases.
A) Formas de representar clases:

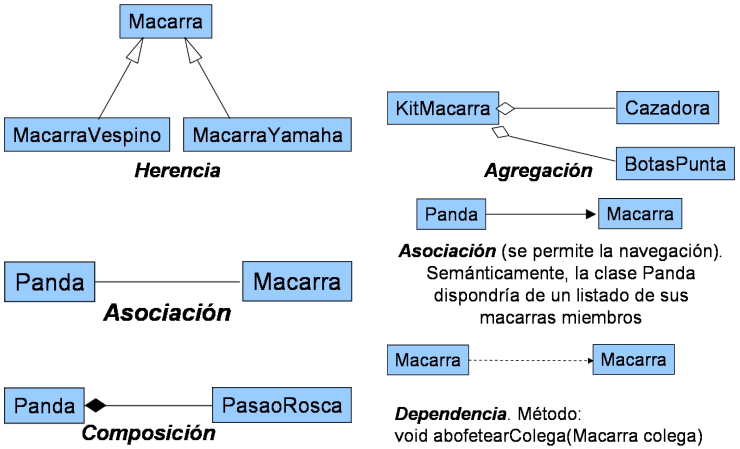


UML

B) Formas de representar objetos:

UML

C) Relaciones entre clases



Contenido del tema

Relaciones entre clases

Motivación:

Problemas reales: diferentes clases cuyos objetos interrelacionan entre sí.

Se repasan aquí las interacciones más frecuentes, tal y como se representan en UML.

Relaciones entre clases

1. Asociación

Es una **relación general binaria** que describe una actividad entre dos clases.

- Por ejemplo, la relación puede expresar que un profesor **enseña** una asignatura, que un estudiante **cursa** una asignatura, que un cliente **vive en** una dirección, que un trabajador **dirige** una empresa, etc.

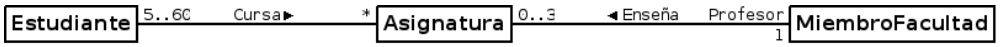
Relaciones entre clases: asociación

- Representación con línea continua con una etiqueta opcional que describe la relación.
- Puede aparecer también un pequeño triángulo negro que describe la dirección de la relación.
- Cada clase de la relación puede mostrar el nombre del **rol** que juega en la relación. (Ej.: Profesor)
- Cada clase de la relación puede especificar una **multiplicidad**, que se coloca al lado de la clase para especificar cuántos objetos de la clase están involucrados en la relación.
  - *: número no limitado de objetos.
  - **m..n**: número de objetos entre **m** y **n**.

Relaciones entre clases: asociación

Implementación de asociaciones en Java

En una **relación bidireccional** añadiremos un dato miembro en cada clase para que un objeto de una clase pueda acceder al objeto u objetos de la otra clase con la que está relacionado.




```
public class Estudiante {
  private Asignatura[]
    listaAsignaturas;
  public void aniadirAsignatura(
    Asignatura a) { ... }
}
```

```
public class Asignatura {
  private Estudiante[]
    listaEstudiantes;
  private MiembroFacultad profesor;
  public void aniadirEstudiante(
    Estudiante e)
  { ... }
  public void ponerProfesor(
    MiembroFacultad profesor)
  { ... }
}
```

```
public class MiembroFacultad {
  private Asignatura[]
    listaAsignaturas;
  public void aniadirAsignatura(
    Asignatura a)
  { ... }
}
```


Relaciones entre clases: asociación

Implementación de asociaciones en Java

En una **relación unidireccional** solo añadiremos el dato miembro en la clase desde la que podemos navegar hacia la otra.


```
public class Estudiante {
  private Asignatura[]
    listaAsignaturas;
  public void aniadirAsignatura(
    Asignatura a) { ... }
}
```

```
public class Asignatura {
  ...
}
```

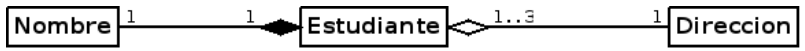
```
public class MiembroFacultad {
  private Asignatura[]
    listaAsignaturas;
  public void aniadirAsignatura(
    Asignatura a)
  { ... }
}
```


Relaciones entre clases: agregación

2. Agregación

Es un tipo de asociación que representa una relación de propiedad entre dos objetos. Una agregación modela la relación **tiene-un**: una clase es el todo y la otra una parte (un objeto de una clase es un componente de un objeto de la otra clase).
Ejemplo: un Estudiante tiene una Direccion.

- El objeto propietario es el objeto agregador y su clase la clase agregadora.
- El objeto contenido se llama objeto agregado y su clase, la clase agregada.
- Representación: línea con punta de rombo (color blanco) en la clase que representa al todo.



Relaciones entre clases: composición

3. Composición

Tipo de relación de agregación en que el objeto agregador es dueño del agregado de forma exclusiva.

Ejemplo: Un Estudiante tiene un Nombre.

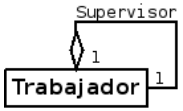
- Por tanto, la parte desaparecerá al desaparecer el todo.
- Representación: línea con punta de rombo (color negro) en la clase que representa al todo.

Relaciones entre clases: relaciones involutivas

Relación involutiva

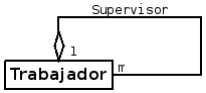
Cuando la misma clase aparece en los dos extremos de la asociación

Ejemplo: un trabajador tiene un supervisor




```
public class Trabajador {
  private Trabajador supervisor;
  ...
}
```

Ejemplo: un trabajador que puede tener varios supervisores

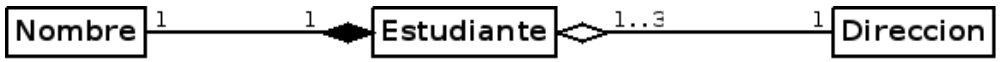



```
public class Trabajador {
  private Trabajador[] supervisores;
  ...
}
```


Relaciones entre clases: asociación

Implementación de agregaciones en Java

Habitualmente una agregación se implementa con un dato miembro en la clase agregadora.




```
public class Nombre {
  ...
}
```

```
public class Estudiante {
  private Nombre nombre;
  private Direccion direccion;
  ...
}
```

```
public class Direccion {
  ...
}
```

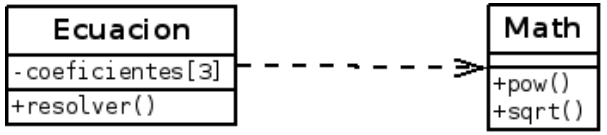

Relaciones entre clases: dependencia

4. Dependencia

Una clase usa objetos de la otra. Muestra la relación entre un cliente y el proveedor de un servicio usado por el cliente.

- Relación más débil que una asociación.
- Representación: línea discontinua que apunta del cliente al proveedor

Ejemplo: Para resolver una ecuación de segundo grado hemos de usar la función `sqrt` de la clase `Math` para calcular la raíz cuadrada.



Relaciones entre clases: herencia

5. Herencia

Una clase deriva (hereda) de otra, denominada clase base. La clase base ofrece todas sus características a la derivada, que puede agregar además sus propios detalles.

Línea continua con punta de flecha triangular (en blanco) apuntando a la clase base