Programación Orientada a Objetos

Tema 2: Elementos Básicos de la Orientación a Objetos

Eduardo Mosqueira Rey

LIDIA
Laboratorio de Investigación y desarrollo en Inteligencia Artificial

Departamento de Computación
Universidade da Coruña, España

Objetivos

- Conocer los elementos fundamentales que forman la base de todo lenguaje orientado a objetos.
- Conocer las características de implementación de clases y objetos en los lenguajes de programación: gestión de memoria, mecanismos de visibilidad, etc.
- Aprender a crear definiciones de clases e instanciar objetos en un lenguaje de programación orientado a objetos como Java.
- Conocer el concepto de metaclase y aprender a como usar las metaclases como mecanismo de reflexión.

Índice

1. Clases
2. Objetos
3. Metaclases

Índice

1. Clases
   – Definición
   – Declaración
   – Ejemplo

Clases
Definición

- **Programa orientado a objetos**
  - Se compone de objetos que se comunican entre si intercambiando mensajes
- **Relación clases - objetos**
  - Similar a la existente entre tipos y variables (una clase es la declaración de un tipo objeto)
  - Las clases se utilizan como plantillas que describen como se construyen ciertos tipos de objetos.
  - Cada vez que se construye un objeto a partir de una clase decimos que estamos ejemplificando (o instanciando) dicha clase (un objeto es una ejemplificación de una clase)
- **Definición de clase abarca:**
  - Definición estructural: en donde se define el estado y el comportamiento que tendrán los objetos de esa clase
  - Propósito de creación de nuevos objetos: en donde define los procedimientos para construir nuevos objetos de esa clase.

Clases
Declaración

- **Declaración**

```
[Modificadores] class Nombre [extends clase][implements interface, ...]
{  // Lista de atributos  // Lista de métodos }
```

- **Modificadores de clase**
  - **public**: permite que la clase sea accesible desde otro paquete
  - **abstract**: Define clases que no pueden instanciarse
  - **final**: Define clases que no pueden extenderse con subclases
    - **Nota: si no se pone ninguno de los modificadores quiere decir que la clase es no-public, no-abstract y no-final. No existen modificadores por defecto**
- **Cláusula extends**
  - **Define la superclase de la clase definida (por defecto Object)**
- **Cláusula implements**
  - **Define los interfaces que implementa la clase**

Clases
Ejemplo

- **Definición de clase**

```
public class Caja
{
  // atributos
  private int value;

  // Métodos
  public Caja ()
  {value = 0;}

  public void setValue (int v)
  {  value = v;  }

  public int getValue ()
  { return value; }
}
```

- **Instanciación de la clase**

```
...

Caja x = new Caja();
x.setValue (7);

...
```

Índice

2. Objetos
   - Identidad
     - Gestión de la memoria
     - Comparación de objetos
   - Estado
     - Especificadores de acceso
     - Modificadores
     - Tiempo de vida, inicialización y ámbito
   - Comportamiento
     - Definición de métodos
     - Tipos de retorno y parámetros
     - Métodos constructores
     - Tipos enumerados
     - Destrucción de objetos
     - Recolección de basura
     - Métodos especiales

Objetos

- **Objeto: elemento <u>identificable</u> que contiene**
  - **Características declarativas**
    - **Utilizadas para determinar el estado del objeto.**
  - **Características procedimentales**
    - **Utilizadas para modelar el comportamiento del objeto.**
  - **Características estructurales**
    - **Utilizadas para organizar los objetos con respecto a otros objetos de su entorno.**

Objetos
Identidad

- **Identidad**
  - Propiedad de un objeto que lo distingue de todos los demás.
  - Esta unicidad se consigue a través de unos identificadores de objetos (OID – object identifiers)
  - Los OIDs son independientes de los atributos que definen el estado del objeto y que normalmente se implementan a través de punteros
  - Generalmente los identificadores de objetos son generados por el sistema y no pueden ser modificados por el usuario.

Objetos
Identidad

- **Gestión de la memoria**
  - **Pila (Stack)**
    - **Estructura en la cual solo se pueden incorporar o eliminar elementos por un solo punto denominado cima.**
    - **Se utiliza para las llamadas a funciones, cuando se llama a una función se reserva espacio en la pila para los valores locales y los parámetros de dicha función.**
  - **Montículo (Heap)**
    - **Parte de la memoria que no está ligada a la llamada de funciones y en el cual la memoria se asigna cuando se solicita expresamente (a través del operador new en Java)**
  - **En OO**
    - **Para una mayor eficiencia la pila almacena tipos básicos cuyo tamaño es conocido en tiempo de compilación (int, float, double, etc.) entre los que se incluye el tipo puntero**
    - **Los objetos se implementan como punteros que residen en la pila pero que apuntan a una zona de memoria del montículo**

Objetos
Identidad

- **Gestión de la memoria**

Identidad de Objetos
Comparación de objetos

- Tipos de comparación:
  - Identidad:
    - Dos objetos son idénticos si y solo si son el mismo objetos (es decir, tienen un mismo OID).
  - Igualdad:
    - Dos objetos se consideran iguales si, a pesar de ser objetos distintos (con distinto OID), sus atributos son idénticos
    - El problema se plantea con los objetos compuestos (objetos que contienen a otros objetos) y que hace que se distingan dos tipos de igualdad: igualdad superficial e igualdad profunda.

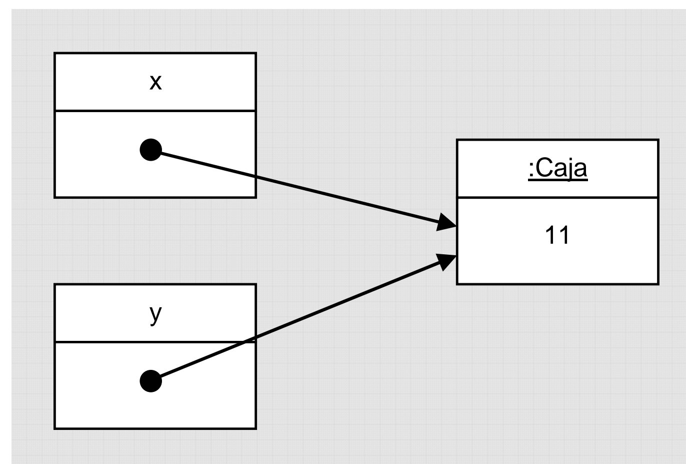
Identidad de Objetos
Comparación de objetos

```
Caja x = new Caja();
x.setValue (7);

Caja y=x;
y.setValue(11);

System.out.println("Valor de x= " + x.getValue());
System.out.println("Valor de y= " + x.getValue());
```

- **Cajas idénticas**



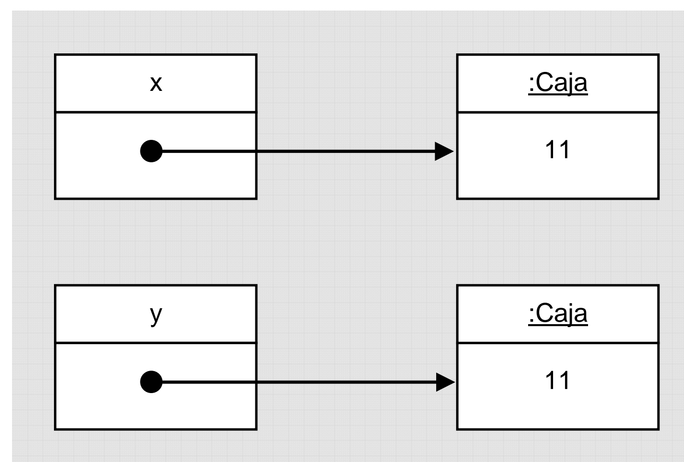
Identidad de Objetos
Comparación de objetos

```
Caja x = new Caja();
x.setValue (11);

Caja y = new Caja();
y.setValue(11);

System.out.println("Valor de x= " + x.getValue());
System.out.println("Valor de y= " + x.getValue());
```

- **Cajas iguales**



Identidad de Objetos
Comparación de objetos

- **Operadores de comparación "==" (identidad) y equals (igualdad)**
  - **¿Cuál es el resultado de las siguientes comparaciones?**

```
Caja x = new Caja();
x.setValue (7);

Caja y = new Caja();
y.setValue(7);

if (x==y) System.out.println ("x e y son idénticos");
else System.out.println ("x e y NO son idénticos");

if (x.equals(y)) System.out.println ("x e y son iguales");
else System.out.println ("x e y NO son iguales");
```

Identidad de Objetos
Comparación de objetos

- **Redefinición del método equals en clases propias**

```
public boolean equals (Object unObjeto)
{
  if (unObjeto instanceof Caja)
  {
    Caja aux = (Caja)unObjeto;
    return this.getValue() == aux.getValue();
  }
  else return false;
}
```

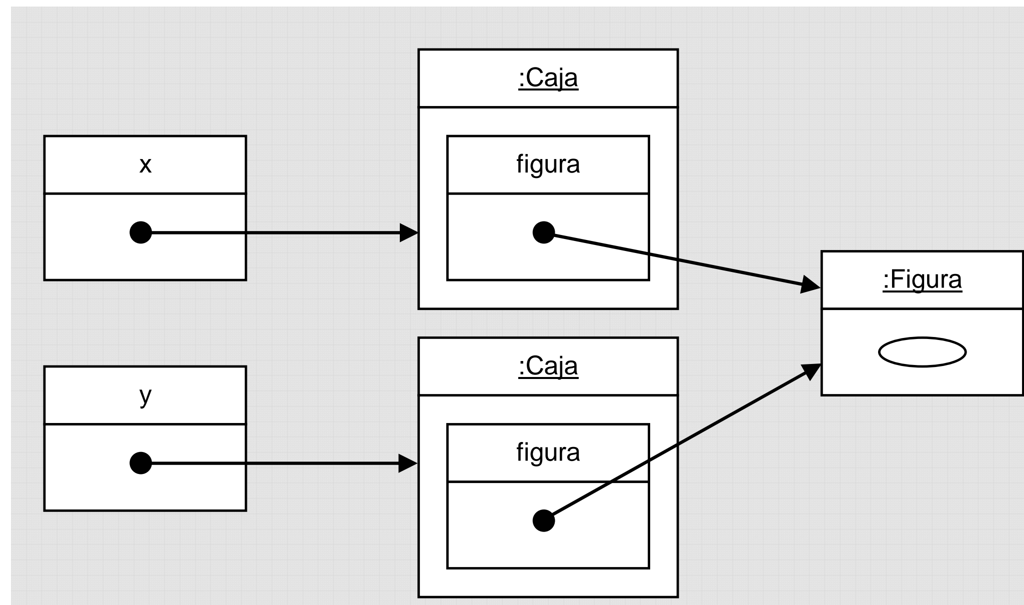
Error típico: poner la siguiente cabecera:
`public boolean equals (Caja unaCaja)`

De esta forma no se sobreescribe el método equals de Object sino que se sobrecarga. Ahora Caja tiene dos métodos equals con funcionamientos dispares

***Downcasting* necesario para acceder al método getValue**

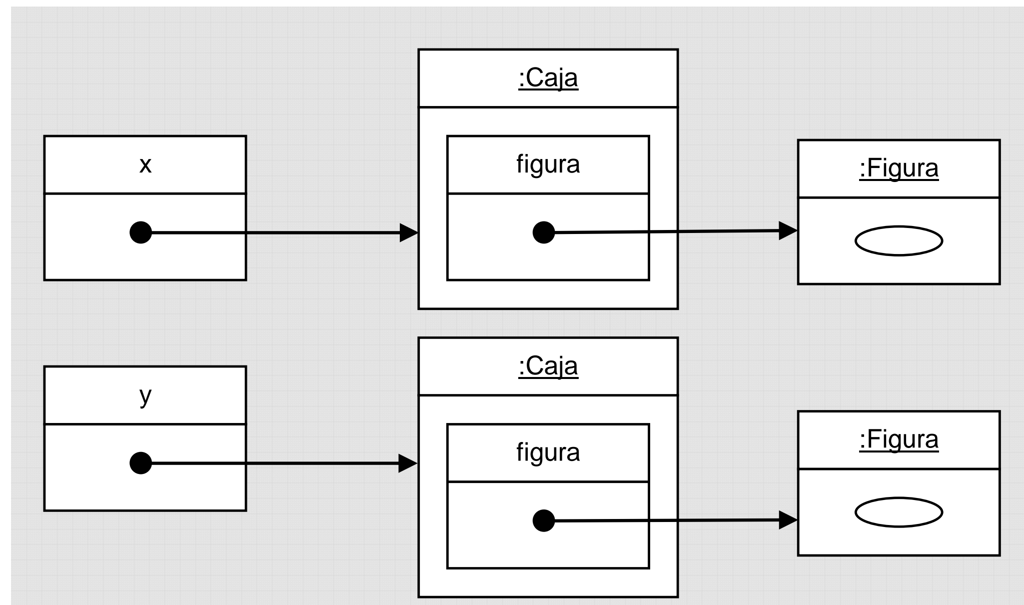
Identidad de Objetos
Comparación de objetos

- **Tipos de igualdad**
  - **<u>Igualdad superficial (shallow)</u>: Dos objetos son superficialmente iguales si los atributos de los objetos son idénticos. Es importante destacar que esta definición no es recursiva.**



Identidad de Objetos
Comparación de objetos

- **Tipos de igualdad**
  - **<u>Igualdad profunda (deep)</u>: Dos objetos son profundamente iguales si sus atributos son recursivamente iguales.**



Identidad de Objetos
Comparación de objetos

- **Clonación de objetos**
  - **En la clase Object existe un método clone que permite realizar copias superficiales**
  - **Debido a que las copias superficiales son peligrosas de manejar se ha protegido la ejecución accidental de este método de varias maneras: tiene visibilidad privada, por lo que hay que redefinirlo con visibilidad pública; y la clase que lo redefina debe implementar el interfaz de marcado Cloneable (que curiosamente no contiene al método clone)**
  - **Puede ser más sencillo clonar los objetos a través de constructores de copia (aunque ahora no podemos ayudarnos del método clone de Object si el objeto no es muy complejo hacer un clon resultará sencillo**

```
public Punto (Punto p)
{
  x=p.getX();
  y=p.getY();
}
```

  - **Recomendación: los objetos siempre deberían saber clonarse a si mismos (no es recomendable que la tarea de clonación se delegue a un objeto externo)**
  - **Ref. sobre el clone y equals: “Effective Java Programming Language Guide” J. Bloch http://developer.java.sun.com/developer/Books/effectivejava/Chapter3.pdf**

Estado de los objetos
Especificadores de acceso

- **Definición**

```
[especificadorAcceso][Modificadores] tipo nombreVariable [= valor_inicial];
```

- **Especificadores de acceso**
  - Indica que elementos son visibles y cuales no

		Nivel de Acceso				
		misma clase	subclase (mismo paquete)	subclase (otro paquete)	otra clase (mismo paquete)	otra clase (otro paquete)
Modificador	public					
	Protected					
	por defecto (paquete)					
	Private					

Estado de los objetos
Especificadores de acceso

- **Public**
  - Los atributos son visibles para todas las clases
- **Por defecto (paquete)**
  - Es el que se aplica si no se especifica ningún especificador
  - Permite que el atributo sea visible a todas las clases que se sitúan en el mismo paquete que la clase original
- **Protected**
  - Permite que el atributo sea visible a las subclases de la clase original
  - También permite el acceso a las clases que se encuentren en el mismo paquete (a diferencia de otros leng. de programación)
  - Si la subclase está en otro paquete sólo se puede acceder a este atributo desde objetos de esa subclase, nunca desde objetos de la superclase
- **Private**
  - Los atributos sólo son visibles dentro de la propia clase
  - El control de acceso se aplica siempre a nivel de clase, no a nivel de objeto. Los métodos de instancia de un objeto de una clase determinada tienen acceso directo a los miembros privados de cualquier otro objeto de la misma clase.

Estado de los objetos
Especificadores de acceso

- **¿Cuál de los siguientes accesos a la variable c son legales?**

```
package paqueteA;

public class Clase
{
  private int privado;
  protected int protegido;
  int paquete;
  public int publico;

  public void metodoDeAcceso()
  {
    Clase c = new Clase();
    c.privado   = 1;
    c.protegido = 2;
    c.paquete   = 3;
    c.publico   = 4;
  }
}

...
```

```
...

class ClaseMismoPaquete
{
  public void metodoDeAcceso()
  {
    Clase c = new Clase();
    c.privado   = 1;
    c.protegido = 2;
    c.paquete   = 3;
    c.publico   = 4;
  }
}

class SubClaseMismoPaquete extends Clase
{
  public void metodoDeAcceso()
  {
    Clase c = new Clase();
    c.privado   = 1;
    c.protegido = 2;
    c.paquete   = 3;
    c.publico   = 4;

    SubClaseMismoPaquete scmp = new
                  SubClaseMismoPaquete();
    scmp.privado   = 1;
    scmp.protegido = 2;
    scmp.paquete   = 3;
    scmp.publico   = 4;
  }
}
```

Estado de los objetos
Especificadores de acceso

- **¿Cuál de los siguientes accesos a la variable c son legales?**

```
package paqueteB;

import paqueteA.*;

class ClaseOtroPaquete
{
  public void metodoDeAcceso()
  {
    Clase c = new Clase();
    c.privado   = 1;
    c.protegido = 2;
    c.paquete   = 3;
    c.publico   = 4;

    Clase c2 = new Clase();
    int i= c2.accesoPrivado(c);
  }
}

...
```

```
...

class SubClaseOtroPaquete extends Clase
{
  public void metodoDeAcceso()
  {
    Clase c = new Clase();
    c.privado   = 1;
    c.protegido = 2;
    c.paquete   = 3;
    c.publico   = 4;

    SubClaseOtroPaquete scop =
            new SubClaseOtroPaquete();
    scop.privado   = 1;
    scop.protegido = 2;
    scop.paquete   = 3;
    scop.publico   = 4;
  }
}
```

Estado de los objetos
Especificadores de acceso

- **¿Cuál es el resultado del siguiente código?**

```
class Clase1
{
  private int valor;

  public int getValor()
  { return valor; }

  public void setValor(int valor)
  { this.valor=valor; }

  public void examinaClase1(Clase1 c1)
  { System.out.println ("Esta Clase1 tiene valor " + c1.valor); }

  public static void main(String[] args)
  {
    Clase1 a = new Clase1();
    Clase1 b = new Clase1();
    a.setValor(5);
    b.examinaClase1(a);
  }
}
```

Estado de los objetos
Especificadores de acceso

- **Ortodoxia de la orientación a objetos**
  - **Los atributos deben declararse privados y su acceso sólo debe ser posible a través de métodos públicos de lectura/escritura.**
  - **Motivos:**
    - **Control de acceso: Se puede controlar el acceso a los atributos (y, por ejemplo, no permitir su modificación si no se cumplen ciertas condiciones)**
    - **Limitar la propagación de modificaciones: Mantener a los atributos abiertos a su acceso directo al exterior complica la posterior modificación de estos atributos. Si el atributo es definido como privado un cambio en sus características sólo afectaría a su propia clase y no a las clases clientes (siempre y cuando no suponga cambios en los métodos de acceso).**
  - **En Java la visibilidad por defecto de los atributos es "package" ⇒ bastante permisiva y, quiza, no la más adecuada**
  - **Hay que tener especial cuidado con los métodos de acceso que devuelven referencias de atributos privados, ya que rompen la encapsulación**

Estado de los objetos
Especificadores de acceso

- **Acceso no deseado a variables privadas**

```
class Cuenta
{
  // Atributos
  private int balance;

  // Métodos constructores
  public Cuenta (int cantidad)
  { balance = cantidad;
  }

  // Métodos de acceso
  public int getBalance()
  { return balance;
  }

  // Otros métodos
  public void retirada(int cantidad)
  { balance = balance - cantidad;
  }

  public void ingreso (int cantidad)
  { balance = balance + cantidad;
  }
}
```

```
class Cliente
{
  // Atributos
  private String nombre;
  private Cuenta cuenta;

  // Métodos constructores
  public Cliente (String nombre, int cantidad)
  {
    this.nombre = nombre;
    cuenta = new Cuenta(cantidad);
  }

  // Métodos de acceso
  public String getNombre()
  { return nombre;
  }

  public Cuenta getCuenta()
  { return cuenta;
  }
}

class PruebaCliente
{
  public static void main (String [] args)
  {
    Cliente cJuan = new Cliente ("Juan", 1000);
    Cuenta c = cJuan.getCuenta();
    c.retirada (1000);
    System.out.println("El saldo de Juan es " +
                       cJuan.getCuenta().getBalance());
  }
}
```

El cliente devuelve una referencia a un elemento privado interno como es la cuenta

Como los objetos son punteros devolver una referencia de un objeto interno privado permite su modificación externa

El caso del atributo nombre es similar pero no ofrece los mismos problemas ¿por qué?

Estado de los objetos
Especificadores de acceso

- **Utilización de clones para evitar el acceso no deseado a variables privadas**

```
class Cuenta implements Cloneable
{
  ...
  public Object clone ()
  {
    try
    {
      Object clon = super.clone();
      return clon;
    }
    catch (CloneNotSupportedException e)
    { return null; }
  }
  ...
}

class Cliente
{
  ...
  public Cuenta getCuenta()
  { return (Cuenta)cuenta.clone(); }
  ...
  public Cuenta(Cuenta c)
  { balance = c.balance; }
}
```

Podemos redefinir con visibilidad pública el método clone de la clase Object (que realiza copias superficiales).

Como las copias superficiales son peligrosas el método clone obliga a implementar el interfaz Cloneable y a lanzar la excepción CloneNotSupportedException para comprobar este hecho

Si el objeto es sencillo puede resultar menos complejo realizar un constructor de copia (aprovechándones de que las restricciones de visibilidad se hacen a nivel de clase

Estado de los objetos
Especificadores de acceso

- **Utilización de objetos inmutables**
  - **Si cuenta fuera un objeto inmutable (no se puede modificar) no existe ningún peligro en compartir referencias)**

```
class Cuenta
{
  // Atributos
  private int balance;

  // Métodos constructores
  public Cuenta (int cantidad)
  { balance = cantidad;
  }

  // Métodos de acceso
  public int getBalance()
  { return balance;
  }
}
```

Si se eliminar los métodos de ingreso y retirada la cuenta, una vez creada, no puede ser modificada

Estado de los objetos
Modificadores

- **Modificadores**
  - static: para definir atributos de clase
  - final: para definir constantes. Pueden inicializarse al definirlas o en el constructor ("blank finals").
  - transient: no es una parte persistente del estado del objeto
  - volatile: es modificado de forma asíncrona

- **Atributos de clase**
  - Un atributo de clase puede ser visto como un atributo que no pertenece a una instancia concreta de una clase sino que pertenece a la clase misma, aunque es accesible por todas las instancias. Es decir, es un atributo compartido por todas las instancias de la clase.
  - Los atributos de clase, al no pertenecer a ninguna instancia en concreto, pueden existir aunque no hayamos instanciado ningún objeto de la clase.

Estado de los objetos
Modificadores

- **Atributos constantes**
  - Se definen a través de la palabra clave final.
  - Nos permiten desarrollar el código de forma más clara ya que evitan el uso de "número mágicos" (es mucho más claro el uso de la constante PI que el uso de su valor numérico).
  - Las constantes además de final suelen declararse como static, pues suelen ser valores que se comparten entre todas las instancias de una clase y, al no variar, no es necesario que cada instancia almacene una copia del valor.
  - Existen ocasiones en las que cada instancia de una clase almacena un valor distinto en una variable constante. Java permite hacer esto a través de los llamados "blank finals", es decir variables que se declaran constantes y a las cuales se les puede dar valor aunque únicamente una vez (normalmente en el constructor de la clase).

Estado de los objetos
Modificadores

- **Ejemplo de atributos de clase y atributos constantes**

```
public class Caja
{
  public static int numCajas = 0;
  public static final double PI=3.1416;
  private final int lado;

  ...

  public Caja (int value, int lado)
  {
    this.value=value;
    this.lado=lado;
    numCajas++;
  }

  public double areaCirculo ()
  { return 4*PI*lado*lado;
  }

  ...
}
```

Estado de los objetos
T. de vida, inicialización y ámbito

- Tipos de variables
  - Atributos de instancia
    - Definen el estado particular de cada instancia de una clase
  - Atributos de clase
    - Todas las instancias de una clase comparten el valor de los mismos
  - Variables locales
    - Variables definidas dentro de un bloque determinado de código
  - Parámetros
    - Variables utilizadas para pasar información a un método

- Características
  - Tiempo de vida
    - Intervalo de tiempo que transcurre desde que se crea la variable hasta que se destruye, y su memoria se libera para otros usos
  - Inicialización
    - Especifica qué valores iniciales se le asignarán a las distintas variables.
  - Ámbito
    - Es la parte del programa en la cual podemos acceder a dicha variable

Estado de los objetos
T. de vida, inicialización y ámbito

	Tiempo de Vida	Inicialización	Ámbito
Atributos de instancia	Desde la creación del objeto al que pertenece hasta la destrucción del mismo	Pueden inicializarse explícitamente, sino se inicializan a cero	Depende del especificador de visibilidad
Atributos de clase	Desde la carga de la clase por la JVM hasta la descarga de la misma	Pueden inicializarse explícitamente, sino se inicializan a cero	Depende del especificador de visibilidad
Variables locales	Durante de la ejecución del bloque en el que es definida	Pueden inicializarse explícitamente, sino quedan sin inicializar	Bloque en el que es definido
Parámetros	Durante la ejecución del método al que pertenece	Los parámetros formales se actualizan con los valores de los parámetros actuales	Método al que pertenece

Comportamiento objetos

Mensajes y métodos

- **Mensajes**
  - Los objetos interactúan entre si mediante el intercambio de mensajes.
  - Los mensajes incluyen una petición para la realización de una acción acompañada de información adicional (argumentos) necesaria para realizar esa acción.
  - Si un objeto acepta un mensaje entonces acepta la responsabilidad de llevar a cabo la acción que indica ese mensaje. Esta acción es llevada a cabo mediante la ejecución de un método.
- **Métodos**
  - El comportamiento de un objeto está determinado por los métodos que implementa como respuesta a los mensajes que recibe.
  - Comúnmente los métodos son el único interfaz mediante el cual podemos comunicarnos con el objeto.

Comportamiento objetos

Mensajes y métodos

- Relación entre mensajes y métodos
  - Mensajes y métodos no son equivalentes ya que ante el mismo mensaje los objetos pueden ejecutar distintos métodos dependiendo de diversos factores.
  - Ejemplos:
    - Es común que existan varios métodos en una clase con el mismo nombre pero distinto número de parámetros. En este caso el mensaje recibido por el objeto será el mismo pero lo que variará será la información adicional que nos sirve para determinar que método debe ejecutarse para responder a ese mensaje.
    - También los objetos responden de forma distinta a mensajes iguales (ej. el mensaje dibujar en la clase Circulo y Cuadrado).
  - Esta situación se puede complicar aún más con la llamada ligadura dinámica que consiste en que la identificación del método con el que responder a un determinado mensaje se haga en tiempo de ejecución y no en tiempo de compilación.

Comportamiento objetos

Definición de métodos

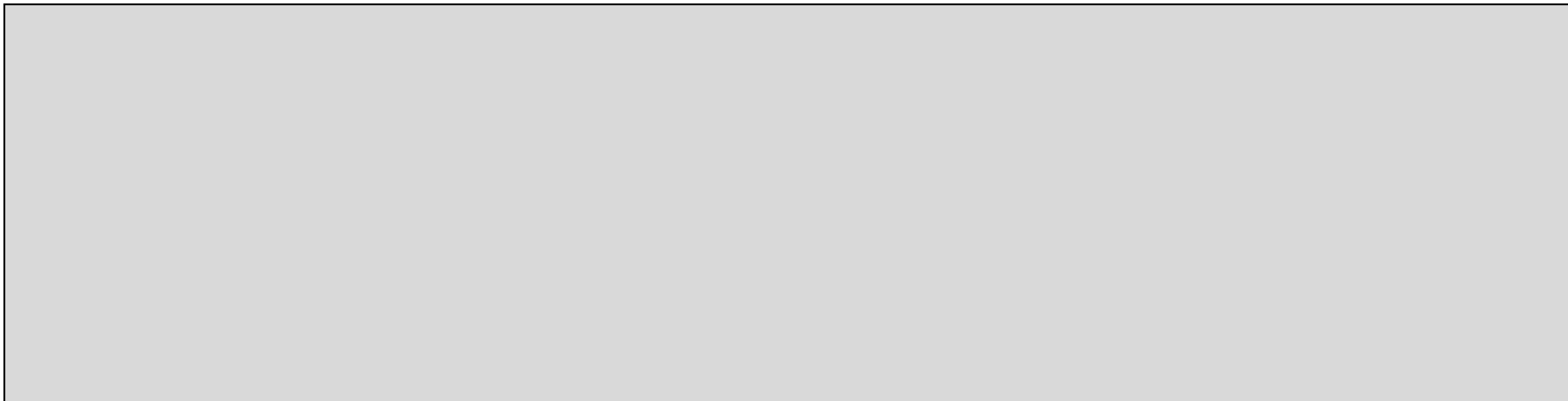
- **Definición**

```
[especificadorAcceso] [Modificadores]
tipoRetorno nombreMetodo([lista_de_parámetros])
      [throws listaEscepciones] { cuerpoMetodo }
```

- **Especificadores de acceso = Especificadores de Atributos**
- **Cláusula throws**
  - Excepciones que puede generar y manipular el método
- **Tipos de Retorno**
  - Los métodos pueden devolver un valor definido (un objeto o un tipo primitivo) o devolver el valor void (vacío) que indica que no ofrecen ningún valor de retorno.
  - Para devolver valores se utiliza la palabra clave return que además provoca la finalización de la ejecución de dicho método.

Comportamiento objetos
Tipos de retorno y parámetros

- **Parámetros formales y actuales**
  - **Parámetros formales**
    - **Son los que se declaran en la implementación del método.**
  - **Parámetros actuales**
    - **Son las variables que se incluyen en la llamada o invocación al método.**
- **En Java los tipos básicos se pasan por valor ¿y los objetos?**



Comportamiento objetos
Tipos de retorno y parámetros

- **Paso de parámetros por valor**

```
// LLAMADA AL MÉTODO
...
metodoA (P1, P2)
...
```

P1	P2
11	● → :ClaseX (...)

Copia de valores de parámetros actuales a parámetros formales

```
// CUERPO DEL MÉTODO

void metodoA (int Param1, ClaseX Param2)
{
...
}
```

Param1	Param2
11	● → :ClaseX

Comportamiento objetos
Tipos de retorno y parámetros

- **Parámetro implícito**
  - Existe un parámetro implícito que es un puntero al objeto al que pertenece el método que se está ejecutando.
  - Este puntero es accesible a través de la palabra clave "this"
- **Ensombrecimiento**
  - Un parámetro formal tiene el mismo nombre que un atributo de instancia.
  - El nombre del parámetro ensombrece al nombre del atributo de instancia (que puede ser accesible a través de "this.nombreAtributo"
- **Devolución de varios valores**
  - Como los parámetros son pasados por valor y los métodos sólo devuelven una determinada variable ¿cómo realizar métodos que devuelvan varios valores?
  - Un truco habitual para solucionar este problema es agruparlos los valores a devolver como atributos de un objeto y devolver una instancia de ese objeto.

Comportamiento objetos
Métodos de lectura y escritura

- **Método de lectura:**
  - Método que accede a un objeto y devuelve diversa información sobre él
- **Método de escritura:**
  - Método que accede a un objeto para modificar su estado
- **Situación típica**
  - Se define una variable de instancia privada y sólo puede ser accedida a través de métodos públicos de la clase.
  - Los métodos de lectura se encargan de devolver el valor de la variable privada mientras que los métodos de escritura se encargan de variar su estado.

Comportamiento objetos
Métodos de lectura y escritura

- **Recomendaciones de uso**
  - Los métodos de lectura sólo devuelven el valor de la variable mientras que los métodos de escritura sólo varían su estado.
  - Si no hay efectos laterales la finalidad de estos métodos es clara
  - Ejemplo (no utilizar println en un método de lectura)

- **Posibles variaciones de funcionamiento admitidas:**
  - Puede existir un método de lectura pero no de escritura (o viceversa)
  - El método de lectura (en caso de tratar con objetos) puede devolver clones de los objetos en vez de los objetos en sí
  - Los métodos de escritura pueden limitar el rango de valores que acepta el atributo.
  - NOTA: En Java los métodos de lectura suelen empezar con el prefijo get y los de escritura con el prefijo set para de esta forma seguir la especificación que indica Sun para los Java Beans, los componentes de Java.

Comportamiento objetos
Modificadores de los métodos

- **Modificadores**
  - static:
    - métodos de clase
  - abstract:
    - métodos no definidos destinados a ser implementados por una subclase
    - Se tratarán con más detalle al hablar de la herencia
  - final:
    - evita que un método sea sobreescrito
    - Si una clase es final todos sus métodos son implícitamente final
    - Mejora la eficiencia del código pero limita su reusabilidad
    - Se tratarán con más detalle al hablar del polimorfismo
  - native
    - Escritos en otro lenguaje (actualmente C y C++)
  - synchronized
    - Usado en la programación multihilo

Comportamiento objetos
Modificadores de los métodos

- **Métodos de clase**
  - Características
    - No pertenecen a un objeto en particular, sino a la clase.
    - Se definen con la palabra clave "static"
    - Pueden ser ejecutados aunque no haya ninguna instancia creada de esa clase: NombreClase.nombremetodo()"
    - No pueden utilizar métodos y atributos no estáticos de la clase (a no ser que lo hagan a través de un objeto), es decir, no existe el puntero this implícito
  - Métodos de clase y orientación a objetos
    - No se corresponden exactamente con la ortodoxia de la programación orientada a objetos, en donde un programa es visto como una serie de objetos que intercambian mensajes
    - Una llamada a un método de clase sería como un mensaje pasado a la clase, no a una instancia en concreto de la clase
    - Como la utilidad de los métodos y atributos de clase está fuera de toda duda, prácticamente todos los lenguajes orientados a objetos los implementan

Comportamiento objetos
Métodos constructores

- **Métodos constructores**
  - Se llaman cuando se instancia un objeto de una clase
  - En Java los métodos constructores se llaman cuando se utiliza el operador new en el código.
  - No hace falta dar una implementación especial al método constructor ya que Java dispone de uno por defecto que lo que hace es inicializar los atributos del objeto (a valores como cero, false, null, etc.).
  - Sin embargo si se quiere realizar una labor concreta en la inicialización del objeto es necesario implementar uno propio (lo que anula la existencia de un constructor por defecto).
  - Características
    - Deben llevar el mismo nombre de la clase
    - No deben devolver ningún valor, ni siquiera void.
    - No estamos limitados a la existencia de un único constructor sino que podemos tener varios siempre y cuando su número de atributos sea distinto.

Comportamiento objetos
Métodos constructores

- ¿Son los métodos constructores métodos estáticos?

- ¿Tiene sentido definir constructores privados?

Comportamiento objetos
Tipos enumerados

- **Los lenguajes generalmente permiten la definición de tipos enumerados**
  - **Pascal**
    - **TColor = (ROJO, VERDE, AZUL)**
  - **C**
    - **typedef enum {POSTRE, ZUMO} naranjas**

- **Diferencias**
  - **En Pascal los enumerado constituían un nuevo tipo que se podía utilizar por ejemplo como índice de un array**
    - **TArray = array [TColor] of integer**
  - **En C es más bien un artificio sintáctico para asignarle un nombre a un entero**
    - **typedef enum {REINETA, GOLDEN} manzanas**
    - **naranjas miNaranja = (GOLDEN – REINETA) / ZUMO; // macedonia**

Comportamiento objetos
Tipos enumerados

- **Enumerados en Java (versión más común)**
  - **La falta de una construcción adecuada para realizar enumerados llevó a construcciones en base a constantes**
  - **Igual que los enumerado de C, no hay ningún control de tipos lo que puede llevar a problemas**

```
class Calificacion
{
  public static final int MATRICULA     = 0;
  public static final int SOBRESALIENTE = 1;
  public static final int NOTABLE       = 2;
  public static final int APROBADO      = 3;
  public static final int SUSPENSO      = 4;
  public static final int NO_PRESENTADO = 5;
}

class PruebaCalificacion
{
  public void introduceCalificacion (int calificacion) { /* ... */ }

  public static void main (String[] args)
  {
    int c = Calificacion.NOTABLE;
    PruebaCalificacion p = new PruebaCalificacion();
    p.introduceCalificacion(c);
    p.introduceCalificacion(287); // Error no detectado
  }
}
```

Comportamiento objetos

Tipos enumerados

- **Enumerados en Java (typesafe enum)**
  - **Patrón “typesafe enum” = enumerados con comprobación de tipos**
  - **Los errores con los enumerados aparecen en tiempo de compilación**

```
class Calificacion
{
 private Calificacion() {};

 public static final Calificacion MATRICULA = new Calificacion();
 public static final Calificacion SOBRESALIENTE = new Calificacion();
 public static final Calificacion NOTABLE = new Calificacion();
 public static final Calificacion APROBADO = new Calificacion();
 public static final Calificacion SUSPENSO = new Calificacion();
 public static final Calificacion NO_PRESENTADO = new Calificacion();
}

class PruebaCalificacion2
{
 public void introduceCalificacion (Calificacion c) { /* ... */ }
 public static void main (String[] args)
   {
     Calificacion c = Calificacion.NOTABLE;
     PruebaCalificacion2 p = new PruebaCalificacion2();
     p.introduceCalificacion(c);
     p.introduceCalificacion(287); // Error en tiempo de compilación
   }
}
```

El constructor se define privado para impedir que se puedan crear instancias desde fuera de la clase

Los enumerados se definen como constantes de clase

Si una función como introduceCalificacion solicita un enumerado, introducir un integer producirá un error en tiempo de compilación

Comportamiento objetos
Tipos enumerados

- **Enumerados en Java (typesafe enum complejo)**
  - **Código repetitivo (“boilerplate”) y propenso a errores**

```
class Calificacion implements Comparable
{
 private int valor;
 private String nombre;

 public int getValor() { return valor; }
 public String toString() { return nombre; }

 private Calificacion(String nombre, int valor)
  { this.nombre = nombre; this.valor = valor; }

 public static final Calificacion MATRICULA = new Calificacion("MATRICULA", 10);
 public static final Calificacion SOBRESALIENTE = new Calificacion("SOBRESALIENTE", 9);
 public static final Calificacion NOTABLE = new Calificacion("NOTABLE", 7);
 public static final Calificacion APROBADO = new Calificacion("APROBADO", 5);
 public static final Calificacion SUSPENSO = new Calificacion("SUSPENSO", 0);
 public static final Calificacion NO_PRESENTADO = new Calificacion("NO_PRESENTADO", 0);

 private static int nextOrdinal = 0;
 private final int ordinal = nextOrdinal++;
 public int compareTo(Object o)
  { return ordinal-((Calificacion)o).ordinal; }

 public static final Calificacion[] VALORES = { MATRICULA, SOBRESALIENTE, NOTABLE,
                                                APROBADO, SUSPENSO, NO_PRESENTADO};
}
```

Cada enumerado lleva asociado ahora una cadena y un entero. Se introducen métodos de lectura de estos valores y se modifica el constructor

Se introduce un atributo ordinal que le asigna un numero de orden a cada enumerado. Esto permite compararlos a través del método compareTo

Se define también un array que contiene a todos los enumerados

Comportamiento objetos
Tipos enumerados

- **Enumerados en Java (Enumerados en Java 5.0)**
  - **Versión simple**

```
enum Calificacion {MATRICULA, SOBRESALIENTE, NOTABLE, APROBADO, SUSPENSO, NO_PRESENTADO};
```

  - **Versión con métodos, atributos y constructores**

```
enum Calificacion
{
 MATRICULA(10),
 SOBRESALIENTE(9),
 NOTABLE(7),
 APROBADO(5),
 SUSPENSO(0),
 NO_PRESENTADO(0);

 private int valor;

 public int getValor()
  { return valor; }

 Calificacion(int valor)
  { this.valor = valor; }
}
```

Comportamiento objetos

Tipos enumerados

- **Enumerados en Java (Enumerados en Java 5.0)**
  - Los enumerados de Java son "syntax sugar", es decir, una comodidad a nivel de sintaxis, ya que el compilador lo que hace internamente es crear una clase como la que hemos visto al describir el typesafe enum
  - Otras ventajas de los enumerados:
    - los valores pueden compararse con == y con equals
    - Implementan el interfaz Comparable por lo que se pueden comparar usando el método CompareTo
    - Sobreescriben el método toString.
      - Calificacion.NOTABLE.toString() devuelve el String "NOTABLE"
    - Tienen un método valueOf que hace el efecto contrario a toString.
      - Calificacion.valueOf("NOTABLE") devuelve Calificacion.NOTABLE
    - Tienen un método ordinal que devuelve el valor ordinal de cada enumerado
    - Tienen un método values() que permite usar los enumerados en un bucle foreach.
      - for (Calificacion c : Calificacion.values()) { ... }

Comportamiento objetos

Destrucción de objetos

- Métodos destructores
  - Definición
    - Se utilizan para eliminar las instancias de los objetos que ya no se utilizan para, de esta forma, liberar la memoria que ocupan
  - Destructores en Java
    - Al contrario que C++ o Object Pascal, no incluye métodos destructores
    - Esto no quiere decir que no haya forma de destruir los objetos sino que esta destrucción se hace de forma automática mediante un proceso denominado recolección de basura
  - Recolección de basura
    - Uno de los principales errores que uno puede provocar a la hora de programar un producto software es el de no liberar correctamente la memoria reservada cuando esta ya no se necesita.
    - El hecho de que los lenguajes incluyan métodos para liberar la memoria no implica que estos métodos sean utilizados correctamente por los programadores (los niños tienen baúles para guardar los juguetes pero generalmente no los usan)

Comportamiento objetos

Destrucción de objetos

- **Recolección de basura**

```
Figura a, b;

a = new Figura();
b = new Figura();

a = b;
```

a	:Figura
b	:Figura

a	:Figura
b	:Figura

recolección de basura

Comportamiento objetos

Métodos especiales

- **Método Main**
  - Cualquier clase (pública o no) puede tener un método main.
  - Si se interpreta un fichero .class, el control se traspasa al método main de la clase (si no existe se produce una excepción).
  - La definición de main siempre tiene que ser:
  - public static void main (String [] args)
  - Hay que tener cuidado con su sintaxis (Java es case-sensitive)

- **Métodos toString**
  - Cuando se espera un String y se pasa un objeto el compilador llama automáticamente al método toString del objeto
  - Los objetos no primitivos de la librería de java tienen definido un método toString. Podemos hacer lo mismo con nuestros objetos si queremos un funcionamiento en concreto a la hora de imprimirlo.
  - Los valores primitivos se convierten automáticamente a string si es necesario

```
public class Caja
{
  private int lado;
  private int value;
  private static int numCajas = 0;
  public static final double PI=3.1416;

  public Caja (int value, int lado)
  {
    this.value=value;
    this.lado=lado;
    numCajas++;
  }

  public Caja ()
  { this (0, 10); }

  public int area ()
  { return lado*lado; }

  public double areaCirculo ()
  { return 4*PI*lado*lado; }

  public void setValue (int value)
  {  this.value = value;  }

  public int getValue ()
  { return value; }

  public static int numeroCajas ()
  { return numCajas; }

  public String toString ()
  { return "Contenido = " + value; }

  public static void main (String [] args)
  {
    Caja unaCaja = new Caja (5, 20);
    Caja otraCaja = new Caja();

    System.out.println ("Area caja 1 : " + unaCaja.area());
    System.out.println ("Area circulo caja 2 : " + otraCaja.areaCirculo());
    System.out.println ("Numero cajas : " + Caja.numeroCajas());
    System.out.println (unaCaja);
  }
}
```

Índice

3. Metaclases
   – Definición
   – Clasificación de sistemas OO
   – Metaclases en Java
   – Ejemplo

Metaclases
Definición

- **Definición**
  - Es la clase de una clase
  - Si un objeto es una instancia de una clase entonces una clase será una instancia de una metaclase.
- **Existencia**
  - En los lenguajes OO sólo se asegura la existencia de objetos y mensajes
  - Las clases no tienen porque existir y mucho menos las metaclases (aunque su uso se recomienda)

Metaclases
Clasificación de sistemas OO

- **Clasificación de sistemas OO según la relación entre objetos, clases y metaclases.**
  - **Sistemas de un nivel**
    - **Sólo hay un tipo de elementos en el sistema, los objetos, no existen ni clases ni metaclases.**
    - **Los nuevos objetos se derivan de los ya existentes**
    - **Lenguajes: Self y Actor**
  - **Sistemas de dos niveles**
    - **Existen dos tipos de elementos en el lenguaje, objetos y clases, siendo los objetos instancias de las clases. No existe soporte de metaclases.**
    - **Lenguaje: C++**
  - **Sistemas de tres niveles**
    - **En estos sistemas todos los objetos son instancias de una clase y todas las clases son instancias de un metaclase. La metaclase también es una clase por lo que es una instancia de si misma**
    - **Existien dos tipos de elementos en el lenguaje: objetos y clases**
    - **Lenguaje: Java**
  - **Sistemas de cinco niveles**
    - **En estos sistemas existen los siguientes cinco niveles: objetos, clases, clases de clases, metaclases y clases de metaclases.**
    - **Aunque realmente existen tres tipos de elementos distintos en el lenguaje: objetos, clases y metaclases.**
    - **Lenguaje: Smalltalk**

Metaclases

Metaclases en Java

- **Java**
  - **En Java existe una clase destacada, la metaclase, cuyas instancias representan a las propias clases de Java.**
  - **Esta metaclase se denomina con el nombre Class, se encuentra en el paquete java.lang y, como todas las clases de Java, extiende a la clase Object.**
  - **La clase Class no tiene métodos constructores sino que las instancias de esta clase son creadas automáticamente por la máquina virtual cada vez que se carga una clase.**
  - **Java permite que los programadores accedan a las instancias de la clase Class y que ejecuten los métodos definidos en dicha clase.**
  - **La principal ventaja de esta accesibilidad es lo que se denomina reflexión, es decir, la posibilidad de que en tiempo de ejecución podamos interrogar a un objeto para obtener la clase a la que pertenece, sus atributos y sus métodos.**
  - **Class también ofrece métodos estáticos muy útiles como forName (que permite obtener el objeto Class de una clase determinada a partir de su nombre**

Ejemplo

```
import java.util.*;         // para acceder a la clase Vector
import java.lang.reflect.*; // para acceder a Field o Method

class reflexion
{
  static void analizaObjeto (Object o)
  {
    Class ClaseObjeto;
    Field[] Atributos;
    Method[] Metodos;
    int i;
    Object NuevoObjeto;

    ClaseObjeto = o.getClass();
    System.out.println("La clase del objeto es ... " + ClaseObjeto.getName());

    Atributos = ClaseObjeto.getDeclaredFields();
    System.out.println("\n" + "ATRIBUTOS");
    for (i=0;i<Atributos.length;i++)
      System.out.println(Atributos[i].getName());

    Metodos = ClaseObjeto.getMethods();
    System.out.println("\n" + "METODOS");
    for (i=0;i<Metodos.length;i++)
      System.out.println(Metodos[i].getName());

    try
    {
      NuevoObjeto = ClaseObjeto.newInstance();
    }
    catch (Exception e)
    {
      System.out.println("Se ha producido un error al crear el objeto");
    }
  }

  public static void main (String [] args)
  {
    Vector miVector = new Vector();
    reflexion.analizaObjeto(miVector);
  }
}
```

Obtenemos el objeto Class mediante el método getClass de Object

Utilizamos el objeto Class para obtener el nombre de la clase, sus atributos y métodos, y crear nuevas instancias

Metaclases
Ejemplo

```
La clase del objeto es ...
java.util.Vector

ATRIBUTOS
elementData
elementCount
capacityIncrement
serialVersionUID

METODOS
wait
wait
wait
getClass
notify
notifyAll
iterator
listIterator
listIterator
hashCode
elementAt
equals
clone
toString
indexOf
indexOf
lastIndexOf
lastIndexOf
get
set
```

```
add
add
addElement
size
toArray
toArray
contains
copyInto
clear
remove
remove
isEmpty
elements
trimToSize
ensureCapacity
setSize
capacity
firstElement
lastElement
setElementAt
removeElementAt
insertElementAt
removeElement
removeAllElements
containsAll
addAll
addAll
removeAll
retainAll
subList
```

Metaclases
Ejemplo

- **Ejemplo de creación de objetos a partir del nombre de la clase**

```
class MetodoForName
{
  public static Object objectForName(String name)
  {
    try
    {
      Class clase = Class.forName(name);
      return clase.newInstance();
    }
    catch (ClassNotFoundException e)
    { System.out.println("Clase no encontrada..."); }
    catch(IllegalAccessException e)
    { System.out.println("La clase no tiene un constructor sin argumentos..."); }
    catch (InstantiationException e)
    { System.out.println("No se ha podido crear el objeto (clase abstracta, interfaz...)"); }
    return null;
  }

  public static void main (String[] args)
  {
    String nombre = "java.util.Date";
    Object o = objectForName(nombre);
    System.out.println("Hemos creado el objeto " + o.toString());
  }
}
```

Creamos un objeto Class a partir del nombre de la clase

Creamos una instancia a partir del objeto Class

Debemos capturar los posibles errores que se puedan producir

Bibliografía

- Bibliografía fundamental
  - Booch, G. “Análisis y diseño orientado a objetos, 2ª ed.”, Addison-Wesley / Díaz de Santos, Wilmington, Delaware, USA, 1996.
  - Arnold, K., Gosling, J., Holmes, D. “El lenguaje de programación Java”, Addison-Wesley, Madrid, 2001.
  - Eckel, B. “Piensa en Java, 2ª ed." Prentice-Hall, Madrid, 2002
  - McLaughlin, B., Flanagan, D. “Java 1.5 Tiger: A Developers Notebook”, O’Reilly, Sebastopol, CA, 2004
- Bibliografía complementaria
  - Campione, M., Walrath, K. “The Java tutorial (2nd edition): Object-oriented programming for the Internet”, Addison-Wesley, Reading, MA, 1998.
  - Horstmann, C. “Computing concepts with Java 2 essentials”, John Wiley and Sons, New York, 2000.
  - Sanchez Allende, J. et al. “Java 2 (2ª ed.)”, McGraw-Hill, Madrid, 2005
- Bibliografía en Internet
  - Ellmer, E. “Basic Principles and Concepts of Object-Orientation (1993)”. URL: http://citeseer.nj.nec.com/ellmer93basic.html
  - Campione, M., Walrath, K. “The Java tutorial (2nd edition): Object-oriented programming for the Internet”. URL: http://java.sun.com/docs/books/tutorial/
  - Eckel, B. “Thinking in Java”. URL: http://www.mindview.net/Books/TIJ/