MI1717

# Introducción a la Programación Orientada a Objetos en Java

[···] netmind

MI1717

# Introducción a la Programación Orientada a Objetos en Java

[···] netmind

# Cómo usar este fichero PDF

## SAVE THE PLANET!

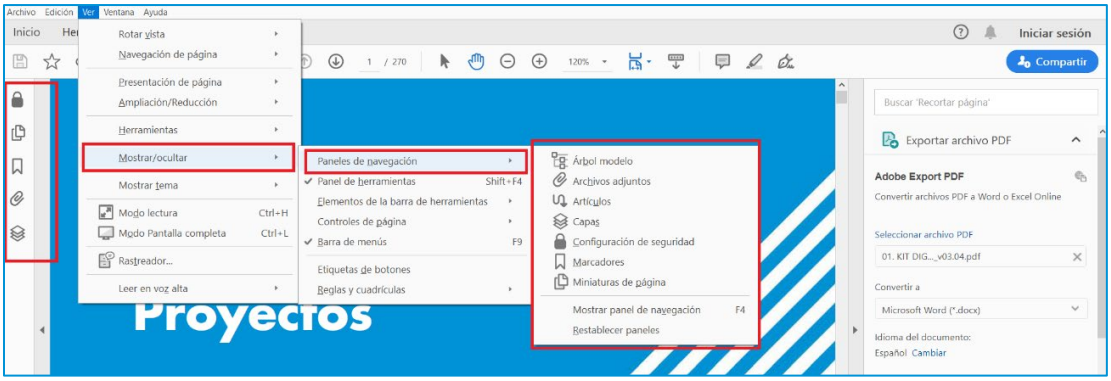
En nuestra lucha por preservar el medio ambiente, hemos deshabilitado las opciones de impresión para que utilices este documento en formato digital en la medida que sea posible.

## PRESERVE THE PRIVACY

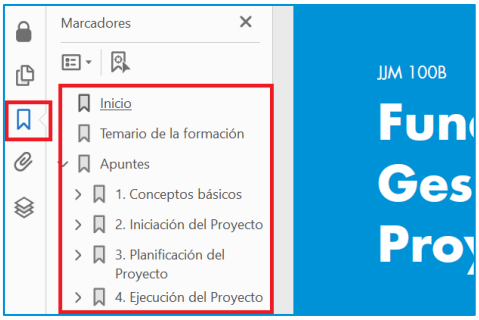
Por cuestiones de confidencialidad, copyright y para evitar modificaciones que alteren su finalidad, la edición de este documento está inhabilitada.

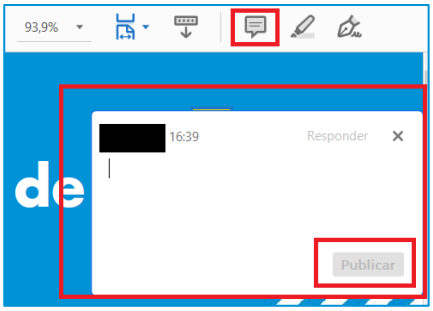
---

- Se recomienda usar el lector de **PDF Adobe Reader** para consultar el archivo.
  - En caso de no tenerlo disponible, puedes obtenerlo AQUÍ.
- Con el lector de Adobe podrás añadir opciones de uso del lector haciendo click en el menú superior **Ver\Mostrar/Ocultar\Paneles de navegación**




- Principales opciones de uso:
  - **Botón "Marcadores".** Se desplegará una lista índice de contenidos que permite navegar por las distintas secciones del documento sin tener que hacer *scroll*.




- **Botón "Comentarios".** Hacemos click en este botón para añadir comentarios o notas relacionadas con el contenido al fichero. Está en el menú superior.

  Se abrirá un cuadro en el que anotar lo que el alumno desee.

  Al finalizar, se hace click al botón "Publicar" y aparecerá un icono de comentario allí donde se haya añadido la nota.

**Para cualquier duda relacionada, contactad con la coordinadora del curso o formacion@netmind.es**

**Barcelona** Carrer Almogàvers 123 (Edificio Ecourban) / T. +34 933 041 720 / F. +34 933 041 722 / 08018 Barcelona
**Madrid** Plaza de Carlos Trías Bertrán 7, 1ª Planta (Edificio Sollube) / T. +34 914 427 703 / 28020 Madrid
**Atlanta** 296 South Main Street, Suite 300/ T. +1 (404) 678 366 1363 / Alpharetta, Atlanta, GA 30009-1973

info@netmind.es | www.netmind.es

# Introducción a la Programación Orientada a Objetos en Java

CODIGO MI1717

Duración: 20 horas

## Introducción

El objetivo del curso es proporcionar a los alumnos las bases teóricas y prácticas necesarias para diseñar y desarrollar aplicaciones siguiendo el paradigma de Programación Orientada a Objetos en Java.

## Objetivos

Al finalizar este curso los alumnos podrán:

- Conocer los principios de la orientación a objetos y como se aplican usando Java.
- Diseñar aplicaciones siguiendo los principios de la Orientación a Objetos.
- Crear código reutilizable usando herencia.
- Utilizar estructuras de datos en Java para almacenar y localizar datos.
- Conocer los principios de la gestión de errores en Java y el uso de excepciones.
- Conocer los casos de uso apropiados para las clases abstractas y las interfaces.

## Dirigido a

Ingenieros y programadores que deseen desarrollar aplicaciones usando Java aplicando los conceptos y principios de la Orientación a Objetos.

## Requisitos

Conocimientos básicos de Java.

## Profesorado

Contamos con un equipo de instructores altamente cualificados que combinan la actividad formativa con el desarrollo de su actividad profesional como expertos en el campo de las TIC. Profesionales certificados por los principales fabricantes del sector capaces de transferir de forma amena y entendedora los conceptos técnicos más abstractos.

## Metodología

Curso online, activo y participativo. El docente introducirá los contenidos haciendo uso del método demostrativo, los participantes asimilarán los conocimientos mediante las prácticas de aplicación real.

## Documentación

Cada alumno recibirá un ejemplar de la Documentación.

## Contenidos

1. Origen y motivación de la POO
   1.1. Abstracción
   1.2. Diferencias entre clase y objeto
   1.3. Estructura de una clase
   1.4. Propiedades y métodos
   1.5. Constructores e instanciación
   1.6. Getters y Setters: Uso de this
   1.7. Miembros static de una clase

2. Introducción al diseño de aplicaciones OO
   2.1. Arquitectura cliente/servidor
   2.2. MVC
   2.3. Ciclo de desarrollo de software
   2.4. Metodologías Ágiles
   2.5. Best Practices

3. Herencia
   3.1. Modificadores de acceso
   3.2. Herencia simple y compuesta
   3.3. Polimorfismo: sobreescritura y sobrecarga
   3.4. Trabajando con referencias de objetos
   3.5. Casting de referencias
   3.6. Clase Object

4. Estructuras de datos
   4.1. Estructuras condicionales
   4.2. Estructuras dinámicas List, Set y Map

4.3. Patrón: Iterator
4.4. Patrón: Observer
4.5. Inferencia de Tipos (Java 10+)

5. Gestión de errores
   5.1. Control de errores, excepciones en POO
   5.2. División de tareas
   5.3. Gestión de responsabilidades
   5.4. Jerarquía de excepciones
   5.5. Estructuras de captura

6. Principios de arquitectura de software
   6.1. Clases abstractas
      6.1.1. Familias de objetos.
      6.1.2. Relaciones de agregación VS composición.
      6.1.3. Diseño de código reutilizable.
   6.2. Interfaces.
      6.2.1. Identificación de comportamientos en objetos.
      6.2.2. Design by contract.

## Evaluación del curso

Evaluación continua en base a las actividades realizadas en grupo y/o individualmente. El formador proporcionará feedback de forma continuada/al final de las actividades/individualmente a cada participante.

En el curso se realizará una prueba de evaluación tipo test que deberá superarse en un 75%. Se dispondrá de una hora para su realización.

## Acreditación

Se emitirá Certificado de Asistencia sólo a los alumnos con una asistencia superior al 75% y Diploma aprovechamiento si superan también la prueba de evaluación.

## Duración del curso

20 horas.

[ıı] netmind

# Apuntes

www.netmind.es

netmind

WeKnowIT

# MI1717 – Parte 1

## Introducción a la Programación Orientada a Objetos en Java

## Conceptos iniciales

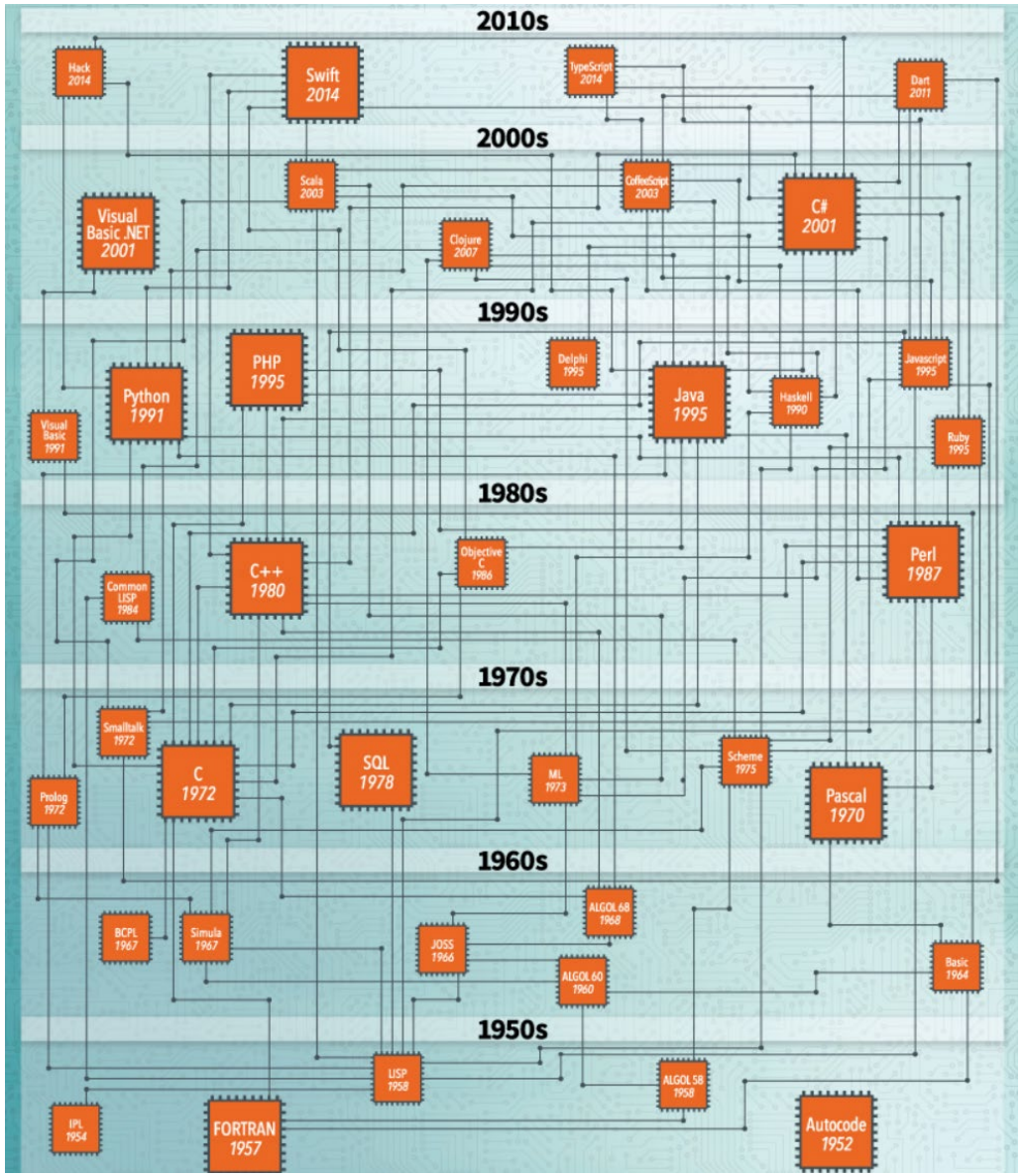
# Programación Orientada a Objetos
# Conceptos iniciales

## Módulo 1

- Evolución de los lenguajes
- Introducción general
- Conceptos básicos
- Diferencias entre clase y objeto
- Composición de una clase
- Estructura de una clase
- Concepto básico de herencia: tipos

# Evolución de los lenguajes

- La evolución de los lenguajes en su historia de vida queda simplificada en la siguiente imagen, que muestra la fecha de aparición con respecto al tiempo:



# Evolución de los lenguajes (y II)

- Si nos centramos en los lenguajes de programación orientados a objetos tendríamos la siguiente lista resumida:
  - 1965: **Simula**
  - 1968: **COBOL**
  - 1980: **Smalltalk**
  - 1984: **Objective-C**
  - 1985: **C++**
  - 1991: **Python**
  - 1995: **Java** y **PHP**
  - 2001: **C#**
  - 2012: **TypeScript**
  - 2014: **Swift**

# Introducción general

## Comparando formas de programar.....

- La programación convencional o estructurada, se concentra en la lista de acciones secuenciales sobre un conjunto de datos, mientras que en la **POO**, **las estructuras son el pivote de la programación**.
- "El término de Programación Orientada a Objetos indica más una **forma de diseño** y una **metodología de desarrollo** de software que un lenguaje de programación."
- "La programación **estructurada** presta atención al **conjunto de acciones que manipulan el flujo de datos** (desde la situación inicial a la final), mientras que la **programación orientada a objetos** presta atención a la **interrelación que existe entre los datos y las acciones** a realizar con ellos."

# Introducción general (II)

## ¿Por qué el uso de la Programación Orientada a Objetos?

- "La orientación a objetos promete mejoras de amplio alcance en la forma de diseño, desarrollo y mantenimiento del software ofreciendo una solución a largo plazo a los problemas que existen en el desarrollo de software:
    - La falta de **portabilidad del código**.
    - **Reusabilidad**, código que es difícil de modificar.
    - **Ciclos** de desarrollo **largos**.
    - Técnicas de **codificación no intuitivas**.

## Ventajas de POO

- Uniformidad y Compresión
- Flexibilidad
- Reusabilidad
- Bajo acoplamiento
- Reducción del ciclo de desarrollo
- Cohesión

# Introducción general (y III)

## Beneficios que se obtienen con el desarrollo de la POO

- Permite obtener **aplicaciones más modificables**, fácilmente extensibles y a partir de componentes reusables.
- Esta **reutilización del código disminuye el tiempo** que se utiliza en el **desarrollo** y hace que éste sea mas intuitivo porque la gente piensa naturalmente en términos de objetos más que en términos de algoritmos de software.
- El esfuerzo del programador ante una aplicación orientada a objetos se centra en **la identificación de las clases, sus atributos y operaciones asociadas**.
- Nos permite **dividir el problema en pequeñas partes** para simplificar el diseño y la implementación o desarrollo del código.

# Conceptos básicos

## Principios de la Programación Orientada a Objeto

- **Abstracción**
- **Encapsulamiento**
- **Cohesión**
- **Herencia**
- **Polimorfismo**

## Características básicas de un lenguaje Orientado a Objetos

- Debe manipular **objetos** basados en **clases** y ser capaz de manejar **herencia** entre clases.

# Diferencias entre clase y objeto

**Clase: Un prototipo o modelo que define las variables y métodos comunes a todos los objetos de un cierto tipo.**

- **Es una plantilla o un molde**

**Características de los objetos**

- Poseen un estado que viene definido por los valores de propiedades específicas.
- Tienen un comportamiento que puede ser modificado utilizando los métodos disponibles.

**Un objeto corresponde a una instancia de una clase pero una clase no es un objeto.**

# Diferencias entre clase y objeto

## Noción de Objeto

- "…son cosas que se pueden percibir por los 5 sentidos.."
- "… conjunto complejo de datos y programas que poseen estructura y forman parte de una organización."
- "…representación real o abstracta del mundo real."
- "…cualquier cosa real o abstracta, en la que se almacenan datos y que contienen métodos que los manipulan".

## Ejemplo de objetos reales:

- **Coche**
- **Reloj**
- **Factura**
- **Cliente…**

# Composición de una clase

**Características de las clases: objetos reales**

- Tienen propiedades específicas
- Tienen un comportamiento
- Relaciones

**Ejemplos:**

- Factura:
  - Propiedades ➔ identificador, importe, estado del pago
  - Comportamiento ➔ anular, eliminar, asignar
- Bicicleta:
  - Propiedades ➔ marca, cadencia de los pedales, velocidades
  - Comportamiento ➔ frenado, acelerado, cambios

# Composición de una clase (II)

## Concepto de clase en Java o C#

- Los métodos son los encargados de utilizar y acceder a las variables de instancia. Una clase se declara con la palabra clave **class**, seguida del nombre de la clase.
- El cuerpo de la clase, está compuesto por miembros (variables de instancia/clase y métodos)

- Una clase se define así:

```
class  [nombre de clase] {
...
}
```

- Si la clase es ejecutable debe contener el método main y su firma es:

```
public static void main (String args[ ]) {
...
}
```

# Composición de una clase (II)

## Concepto de clase en Java o C#

- El método **main()** indica a que la clase es ejecutable.

- Las llaves de la clase **{ }**, sirven para definir el contenido de la clase y puede contener métodos, variables o bloques de código.

- El paquete básico que cada clase debe importar es **java.lang** (en Java) y **System** (en .NET). Pero no es necesario indicarlo explícitamente, ya que el compilador lo hace por nosotros.

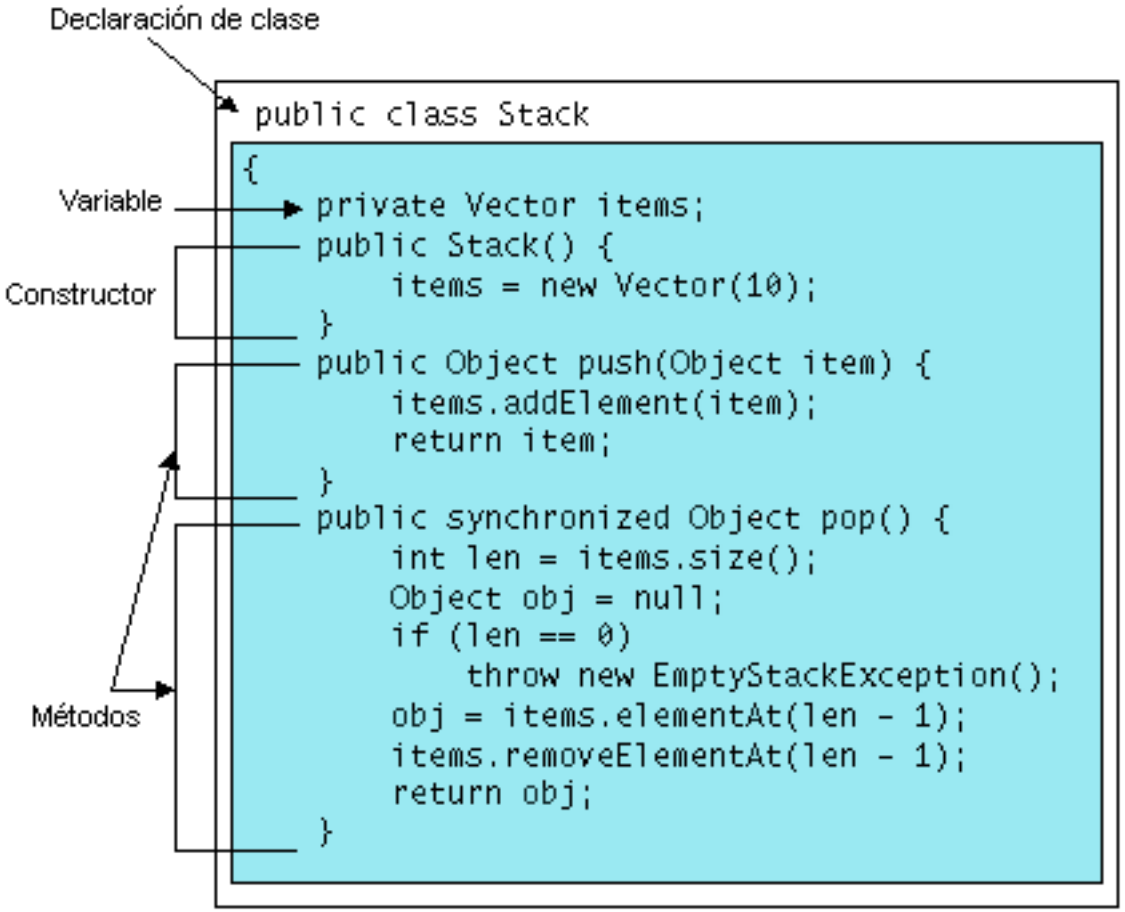
- **Una clase es un patrón que nos puede servir de "molde" para crear otras clases a partir de ella.** A partir de las clases obtenemos objetos (instanciamos) en Java. **Las clases las agrupamos por tareas afines en paquetes.** Existen diversos tipos de clases: normales, internas, abstractas…

# Estructura de una clase

## Estructura básica de una clase

- Una clase POO, no es un objeto. Sirve de plantilla para crear objetos a partir de ella.

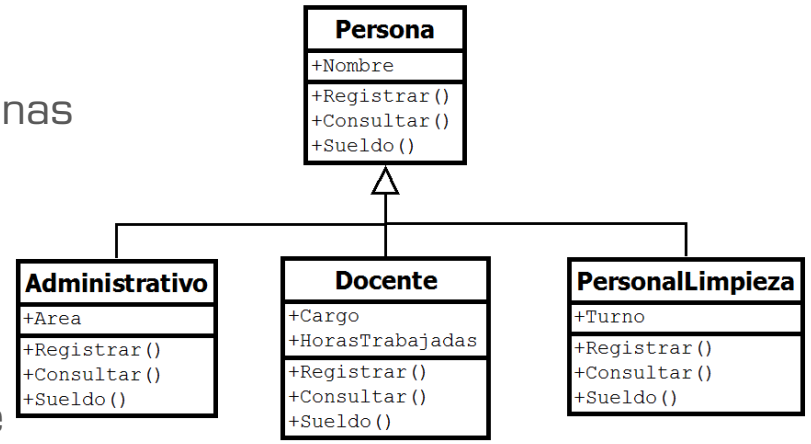
- Una clase puede estar compuesta por:
  - **Variables**
  - **Bloques de código**
  - **Métodos**
  - **Clases internas**

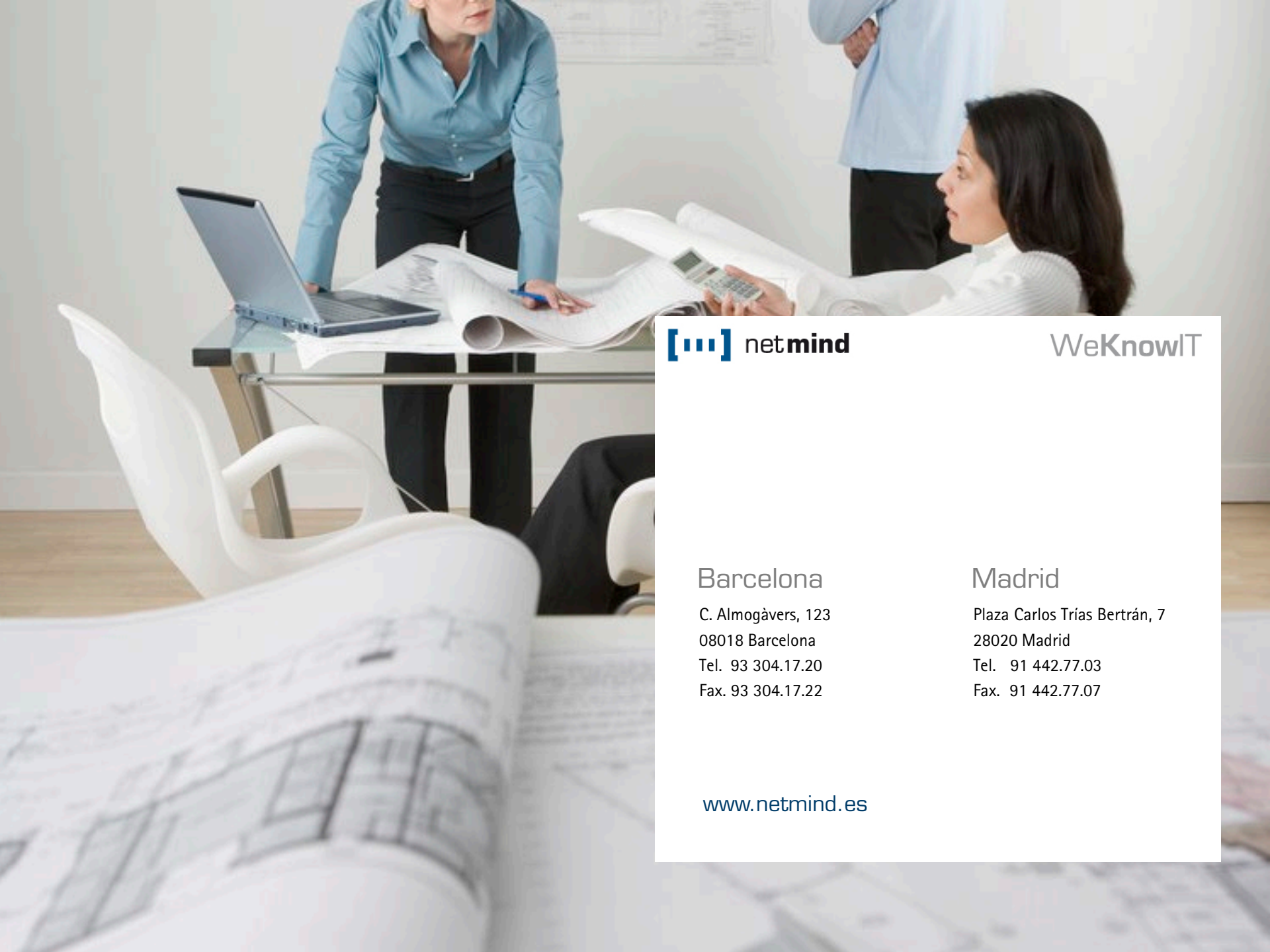


# Concepto básico de herencia: tipos

## Herencia

- Es una característica de los lenguajes POO. Se da soporte a la herencia simple usando clases y la herencia múltiple controlada a través de interfaces.
- La herencia es un mecanismo que nos permite hacer una alta reutilización de clases, estableciendo una plantilla común a partir de la cual podemos obtener objetos de múltiples tipos y variaciones. Como contrapartida se trata de un mecanismo que acopla de manera muy alta las clases que hacen uso del mismo.
- El acoplamiento define la dependencia de unas clases con otras. Cuanto mayor es esta dependencia más baja es la flexibilidad.
  Es necesario hacer un uso adecuado de la herencia para permitir que la flexibilidad de una aplicación sea alta.





netmind

WeKnowIT


## Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

## Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

[···] netmind

WeKnowIT

# MI1717 – Parte 2

## Introducción a la Programación Orientada a Objetos en Java

## **Utilizando objetos**

# Programación Orientada a Objetos
# Utilizando objetos

## Módulo 2

- Introducción
- Tipos de datos y variables
- Conversiones permitidas
- Elementos estáticos de clase
- Inicialización de objetos
- Constructores
- Polimorfismo
- Instanciación

# Introducción

- **Los objetos**, son modelados a partir de los objetos reales, basándose en su **estado y comportamiento**.

  Las propiedades o **atributos** de un objeto se almacenan en:

  - **VARIABLES**

- Los comportamientos se implementan utilizando los:

  - **MÉTODOS**

- **Variable**: es un elemento de datos referenciado por un identificador.
- **Método**: es una función o procedimiento asociado a un objeto.

  Corresponde con las operaciones que pueden realizarse sobre el objeto.

# Introducción

- **Un Objeto** …, Una **unidad** de software formada por variables y sus métodos asociados.
- Podemos representar **objetos reales** mediante objetos programados
- También podemos representar **objetos abstractos** mediante objetos programados.

- **RELACIONES**

- Las relaciones entre objetos son, precisamente, los enlaces que permiten a un objeto relacionarse con aquellos que forman parte de la misma organización.

# Introducción

## Comunicación basada en mensajes

Podemos pensar en el concepto de mensaje como una unidad de información que se envía a un objeto para que la procese con un fin determinado, ya que:

- **Un objeto solo, no es muy útil….**
- **Un objeto normalmente forma parte de un programa que contiene muchos otros objetos.**
- **A través de la interacción de esos objetos, los programadores alcanzan un alto nivel de funcionalidad y comportamientos mas complejos.**
- **Los objetos interactúan mediante el envío de mensajes.**

# Introducción

## Comunicación basada en mensajes

- Instrucción que se envía a un objeto.
- El objeto al cual se le envía el mensaje.
- El método que se desea ejecutar.
  Cualquier otra información que necesite el método para poder actuar (parámetros).
- Para enviar un mensaje o modificar el estado de un objeto se utiliza la notación del punto, como: **Objeto.metodo(parámetros)**

**Ejemplo:** Telefono.llamar(666555444)

# Introducción

## Abstracción

- Mientras que un objeto es una entidad concreta que existe en tiempo y espacio, una clase representa sólo una **abstracción**, la esencia del objeto.
- Una clase es un **conjunto de objetos** que comparten una estructura común y un comportamiento común.
- Un objeto es simplemente la **instancia** de una clase.
- Cuando se crea una clase se involucran dos procesos:
  - La definición de **atributos** que se utilizarán para almacenar la data de un objeto.
  - La definición de los **mensajes** que se desea que los objetos entiendan. Para cada mensaje se crea un método.

# Tipos de datos y variables

## Variables

- Los identificadores en Java y .NET, permiten dar nombre a las variables, métodos clases y objetos.
- Existe una regla que hay que seguir: "**Pueden comenzar por una letra, el símbolo '$' o el '_' y no hay una longitud máxima**".
- Los nombres de clase comienzan en Mayúsculas, mientras que las variables y los métodos lo hacen en minúsculas (en .NET los métodos comienzan en mayúsculas).
- Si el nombre **está compuesto por varias palabras la inicial de cada una de ellas también se escribe en Mayúsculas**.
- **Una constante se declara con todas sus letras en mayúsculas**, como por ejemplo: IVA.

# Tipos de datos y variables

## Tipos de datos

- Cuando se realiza una operación numérica, el operando más pequeño se "convierte" al mayor, siguiendo estas reglas:
    1. Si un operando es **double**, el otro se convierte a **double**.
    2. Si un operando es **float**, el otro se convierte a **float**.
    3. Si un operando es **long**, el otro se convierte a **long**.
    4. En el resto de casos, ambos se convierten a **int**.
- Por defecto cualquier cantidad sin fracción decimal es considerada por el lenguaje como de tipo **int**.
- Por defecto cualquier cantidad con decimales es considerada **double**.

# Tipos de datos y variables

## Tipos de datos

- Es importante, que en estas ocasiones comprobemos que la variable es del tipo adecuado y no se produzcan los siguientes errores:

```
byte = int + byte;          // debe ser de tipo int
int = float + int;          // debe ser de tipo float
long = float + long;        // debe ser de tipo float
float = double + float;     // debe ser de tipo double
```

- También podemos utilizar el mecanismo de boxing y unboxing si usamos tipos de datos wrapper del lenguaje.

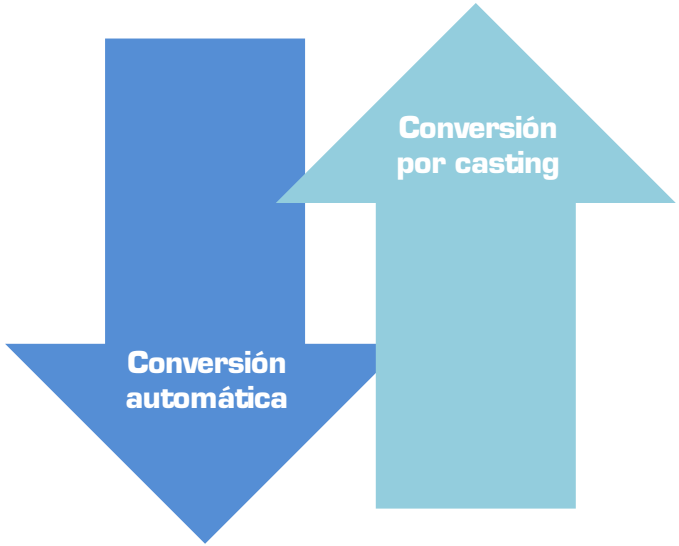
```
int  var1 = new Integer(123);
int var2 = new Integer(24) + 13;
Integer var3 = 124;
```

# Conversiones permitidas

## Tipos de datos

- Java y .NET utilizan dos tipos de datos bien diferenciados: **los primitivos y los wrappers** (envolventes). Los primitivos son los únicos elementos del lenguaje que no son objetos, mientras que los wrapper si lo son.

| Primitivos | Tamaño | Wrapper | Tamaño |
|---|---|---|---|
| byte | 8 bits | Byte | 8 bits |
| short | 16 bits | Short | 16 bits |
| int | 32 bits | Integer | 32 bits |
| long | 64 bits | Long | 64 bits |
| float | 32 bits | Float | 32 bits |
| double | 64 bits | Double | 64 bits |
| char | 16 bits | Character/ Char | 16 bits |
| boolean | 2 valores | Boolean | 2 valores |



# Conversiones permitidas

## Tipos de datos

- Al asignar una variable de un tipo de dato a otro puede ocurrir lo que se conoce como Conversión automática o widening conversion.
- Para ello, tienen que darse dos circunstancias:
    1. Los dos tipos de datos deben ser compatibles
    2. El tipo de dato destino, debe ser más amplio que el de origen.
- La conversión de tipos de dato es automática cuando se convierte el tipo de dato a uno más amplio:

    **byte → short → int → long → float → double**

    **char → int → long → float → double**

- No se permite la conversión de byte y short a char.

# Conversiones permitidas

## Tipos de datos

- Existe otro tipo de conversión, que la que se produce al convertir un tipo de dato "amplio" a otro más "estrecho", se conoce como **narrowing conversion**.
- La regla es la siguiente:

**double → float → long → int → char → short → byte**

- Este tipo de conversión no es implícita y es necesario hacer un casting de forma manual, por ejemplo:

```
int var1 = (int)12.3;
```

- Cualquier tipo de dato puede ser convertido a String.

# Elementos estáticos de clase

## Tipos de variables

- Existen tres tipos básicos de variables:
    1. **Instancia (field o member variable)**
    2. **Clase, marcadas con el modificador static**
    3. **Locales (automatic variable)**
- **Se diferencian por su visibilidad** (scope). En Java o C# hay que especificar obligatoriamente el tipo de dato de la variable.
- **La vida o visibilidad de una variable** (scope) está definida por el bloque que la encierra (es decir las llaves).
- **Las variables pueden inicializarse dinámicamente**. Las variables de instancia se inicializan por defecto con respecto al tipo de dato, **¡ las locales no !**.

# Inicialización de objetos

## Tipos de variables

- Al inicializar una variable sin definir un valor, **Java o C# asignan un valor por defecto según el tipo de variable** o al crear un Array de valores, los valores son los siguientes:

| Tipo de dato | Valor por defecto |
|---|---|
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0 |
| float | 0.0 |
| double | 0.0 |
| char | /u0000 |
| boolean | false |
| Un objeto | null |

# Inicialización de objetos

## Tipos de datos

- Podemos declarar e inicializar estos tipos de datos primitivos según los ejemplos de la siguiente tabla:

| Primitivos | Inicialización |
|---|---|
| byte | byte b1 = 127; |
| short | short s1 = 2345; |
| int | int var1 = 1235; |
| long | long var2 = 250; |
| float | float var3 = 25.3F; |
| double | double var4 = 25.36; |
| char | char var5 = 'm'; |
| boolean | boolean var6 = true; |

# Constructores

- Un constructor **sirve para inicializar las variables** a valores determinados.
- **Si no existe un constructor por defecto, Java o C# crea uno por nosotros**, si existe cualquier otro constructor, no lo hace.
- **El constructor es llamado bajo dos situaciones: Al heredar de la clase y al instanciarla**. Se procesan en primer lugar los constructores de las superclases.
- Sólo **los constructores sin parámetros son implícitamente llamados** al instanciar una clase por defecto.
- Un constructor puede usar las palabras **this** y **super**:
    - **Sólo puede haber una de las dos**
    - **Deben estar situadas en la primera línea del constructor.**

# Polimorfismo

## Polimorfismo en los objetos

- Una de las características fundamentales de la POO es el polimorfismo, que es **la posibilidad de construir varios métodos con el mismo nombre**, pero con relación a la clase a la que pertenece cada uno y con comportamientos diferentes.
- Esto conlleva la habilidad de enviar un mismo mensaje a objetos de clases diferentes.
- Estos objetos recibirían el mismo mensaje global pero responderían a él de formas diferentes; un mensaje "+" a un objeto ENTERO significa suma, pero a un objeto STRING significa concatenación.
- Esta **adaptación al tipo de mensaje** que se recibe se realiza en tiempo de ejecución.

# Polimorfismo (II)

## Conceptos clave

- Podemos definirlo como la capacidad que tienen los objetos para **responder de distinta forma a las invocaciones** de ciertos métodos de una clase, según la estructura del mensaje.
- Existen **dos formas de polimorfismo: la sobrecarga y la sobre escritura**:
- La **sobrecarga** de métodos (**overloading**) es un mecanismo disponible en Java y .NET para implementar polimorfismo y se basa en que varios métodos comparten el mismo nombre, pero con distintos argumentos.
- La **sobre escritura de métodos** (**overriding**) es un mecanismo más "refinado" en la orientación a objetos y se basa en la redefinición de los métodos de la superclase.

# Polimorfismo (III)

## Sobrecarga de métodos

- Para poder trabajar con **la sobrecarga** de se tienen que tener en cuenta estas características:
    - **Aparecen en la misma clase o subclase.**
    - **Tienen el mismo nombre que el "original".**
    - **Tienen una lista de parámetros diferente.**
    - **Pueden devolver diferente tipo de datos.**
- En Java existen muchos métodos sobrecargados como:
    - **Math.abs(double);**
    - **Math.abs(float);**
    - **Math.abs(int);**
    - **Math.abs(long);**

# Polimorfismo (IV)

## Sobrecarga de métodos

- **La llamada al método depende de los parámetros**, Java y .NET de forma automática seleccionan el método adecuado en función de los parámetros recibidos.
- Un método sobrecargado es **diferente → lista de parámetros está en otro orden o estos son diferentes**.
- **Pueden sobrecargase métodos heredados**.
- **Es posible sobrecargar constructores** dentro de la misma clase.
- La sobrecarga de constructores, es la práctica habitual en Java y en C#.

# Polimorfismo (IV)

## Sobrescritura de métodos

- La sobre escritura permite a una subclase "reescribir" un método heredado. Sus características son las siguientes:
  - **Existen en las subclases.**
  - **Poseen el mismo nombre que el método heredado.**
  - **Tienen la misma lista de parámetros.**
  - **Devuelven el mismo tipo de dato.**
  - **Poseen el mismo modificador de acceso.**
  - **Si usan una cláusula throws, sólo pueden incluir excepciones lanzadas por el método de su superclase.**
- En una clase **se pueden sobrecargar múltiples métodos** pero sólo se puede sobrescribir uno.

# Polimorfismo (y V)

## Sobrescritura de métodos

- Un método sobrescrito, no puede tener menos privilegios que el método de su superclase:

```
class Origen {
        void prueba() {
        System.out.println("¡ Comprobación !");
} }
```

- En una subclase:

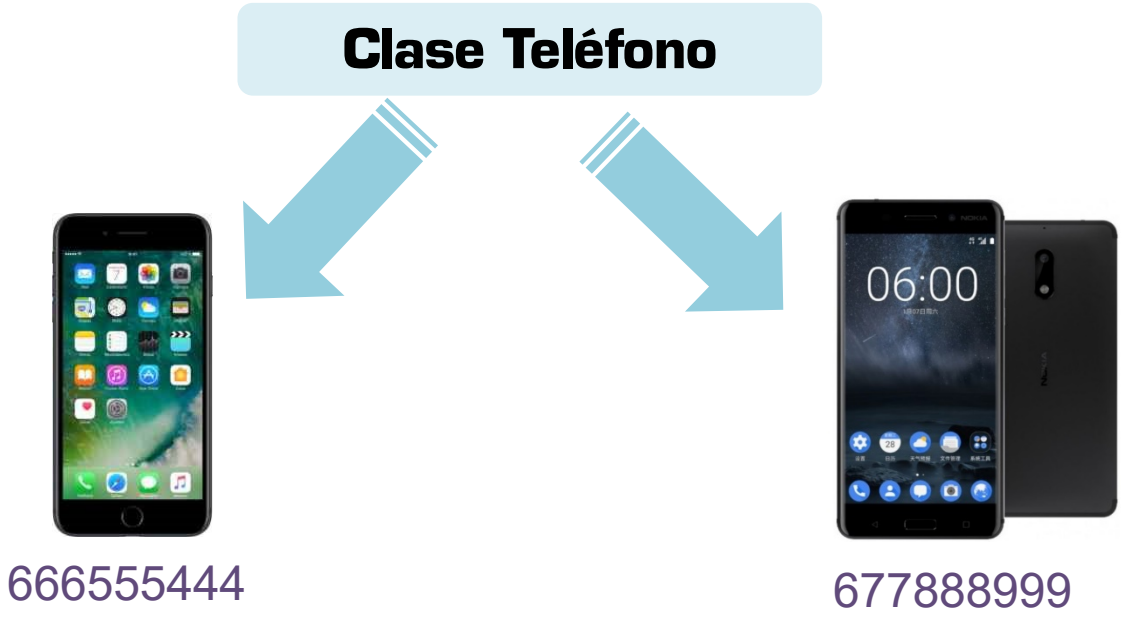
```
private void prueba() {}          // compila bien.
protected void prueba() {}        // compila bien.
```

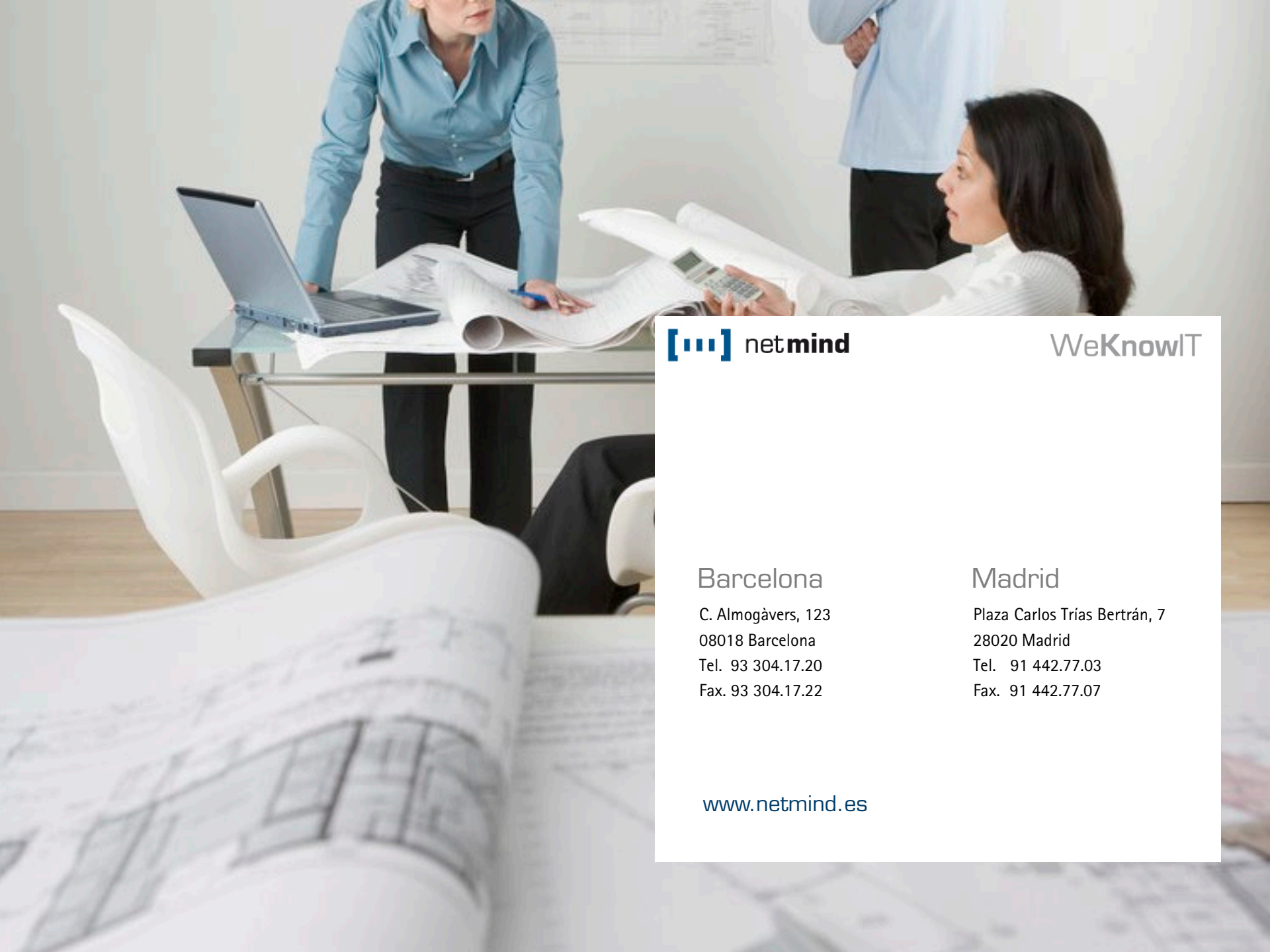
Sólo los métodos no **static** pueden ser sobrescritos. Tampoco es posible sobrescribir un constructor de una superclase.

# Instanciación

## Instancias

- Las instancias de una clase son aquellos objetos de esa clase, que aunque tienen las mismas características tienen valores asociados diferentes para esas características.





netmind

WeKnowIT


## Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

## Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

netmind

WeKnowIT

# MI1717 – Parte 3

## Introducción a la Programación Orientada a Objetos en Java

## **Desarrollo de métodos**

# Programación Orientada a Objetos
# Desarrollo de métodos

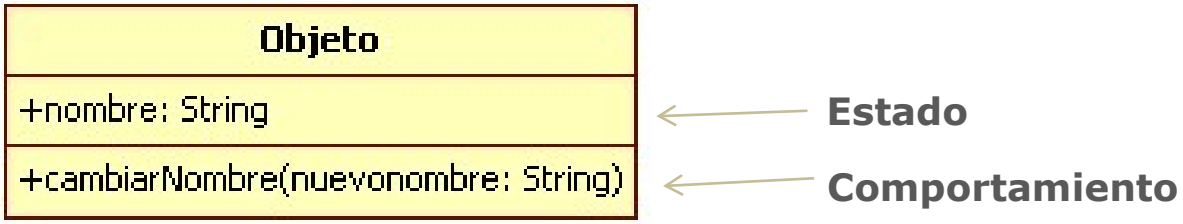
## Módulo 3

- Características de un método
- Argumentos
- Tipos de retorno de datos
- Visibilidad
- Sombreado de variables
- Métodos de clase
- Bloques de código

# Características de un método

## Programación estructurada vs POO

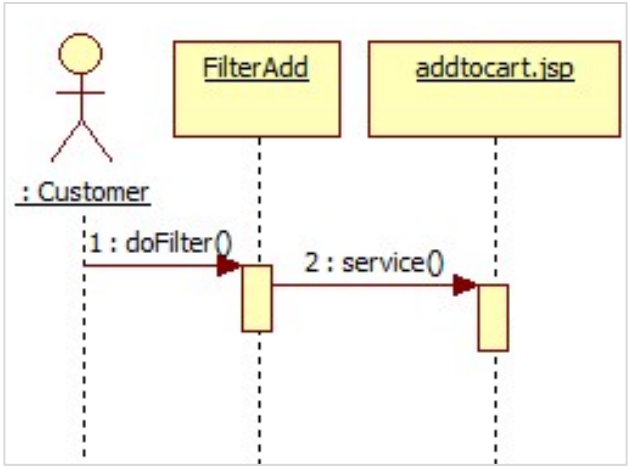
- Los objetos, son modelados a partir de los objetos reales, basándose en sus propiedades y comportamientos. Las propiedades o atributos de un objeto se almacenan en variables o propiedades.
- Los **comportamientos** se implementan utilizando los **métodos**.
- Variable: es un elemento de información referenciado por un identificador.
- **Método**: es una **función o procedimiento asociado a un objeto**. Corresponde con las operaciones que pueden realizarse sobre el objeto.



# Características de un método

## Mensajes entre objetos

- **Un mensaje es una instrucción que se envía a un objeto usando un método**. Un objeto sólo no es muy útil. Un objeto normalmente forma parte de un programa que contiene muchos otros objetos.
- A través de la interacción de esos objetos, los programadores alcanzan un **alto nivel de funcionalidad** y pueden crear comportamientos mas complejos.
- Los objetos interactúan entre sí mediante el envío de mensajes **invocando métodos**.



# Argumentos

## Argumentos de los métodos

- La firma del método nos da información acerca del mismo, por ejemplo su nivel de visibilidad o seguridad, el tipo retorno de datos que tiene (si existe) su nombre y los tipos de datos que admite como argumentos:

**public double calcularIVA(double totalPedido)**

- Con el modificador **public**, nos indica que tiene visibilidad total, **double** a su vez sirve para conocer que devuelve un valor de ese tipo de dato, y finalmente **double totalPedido** que es necesario pasarle un valor de tipo **double** para calcular el porcentaje de IVA que debe ser aplicado.
- Los argumentos son variables de tipo **local/automático**.

# Tipos de retorno de datos

## Retorno de datos de los métodos

- Un método puede devolver valores primitivos (Java) y cualquier tipo de objeto estándar o personalizado (C#, Java y otros), también podemos indicar que no se devuelve nada con el modificador **void**.
- Se utiliza la palabra **return**, que le indica al compilador que se va a devolver un valor.
- Debemos asegurarnos de dos cosas al trabajar con retornos de datos:
    - **Que en cualquier caso se producirá la devolución.**
    - **Que el valor corresponde con el tipo de dato marcado en la firma del método.**

# Visibilidad

## Modificadores utilizados en las clases

- La visibilidad de clases, métodos y variables viene definido en el caso de las clase y métodos por los modificadores de acceso.
- En el caso de las variables por el lugar donde son declaradas:
    - **A nivel de clase: de instancia o de clase**
    - **A nivel de método: locales o automáticas**
    - **A nivel de bloque de código: usando las llaves**
- Además las variables comparten con las clases y métodos el uso de los modificadores de acceso.

# Sombreado de variables

## Duplicidad de variables

- En algunas ocasiones ya que el lenguaje lo permite, **podemos crear variables locales o automáticas que tienen el mismo nombre** que las de instancia o clase.
- En estas ocasiones, se produce el efecto de sombreado de la variable de instancia o de clase, **esta práctica está desaconsejada**:

Código fuente: Ejemplo.java

```
public class Ejemplo {
    int total = 123;
    public static void main(String[] args){
        int total = 555;
        System.out.print("Total: " + total);
    }
}
```

# Métodos de clase

## Métodos static

- Consideramos un método de clase a aquél que ha sido marcado con la palabra **static. Un método static pertenece a la clase y no a las instancias** que podamos obtener de ella.
- Es decir, nos aseguramos de que siempre tendrá el mismo comportamiento sea llamado desde cualquier objeto o instancia.
- **La forma de invocarlo es diferente** y se hace usando la clase directamente, por ejemplo la clase **Math**, dispone de un buen número de ellos

Código fuente

```
Math.abs(2.56);
```

# Métodos de clase (II)

## Métodos de la clase Math

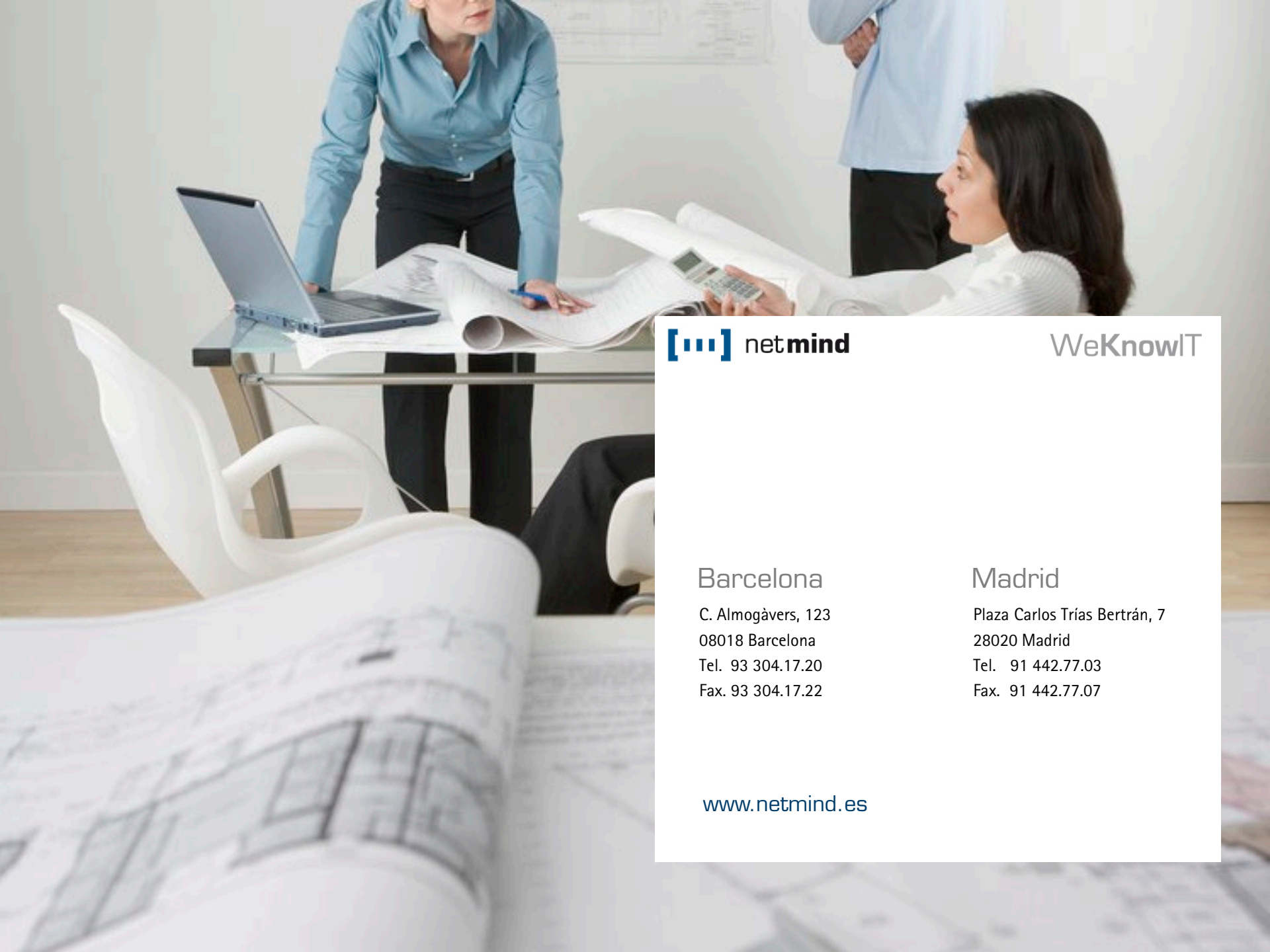
- La clase Math contiene métodos que permiten realizar operaciones básicas numéricas de manera rápida y sencilla, es final y sus métodos **static**:

| Método | Descripción |
|---|---|
| **Propiedades** | El número E y la constante PI |
| **abs()** | Devuelve el valor absoluto de un valor |
| **ceil()** | Devuelve el entero más pequeño y cercano al argumento |
| **floor()** | Devuelve el entero más grande y cercano al argumento |
| **max()** | Devuelve el mayor de dos valores |
| **min()** | Devuelve el menor de dos valores. |
| **random()** | Devuelve un valor double entre 0.0 y 1.0 |
| **round()** | Devuelve el valor más cercano al argumento |
| **sqrt()** | Devuelve el valor redondeado más cercano a la raíz cuadrada del argumento |

# Bloques de código

## Concepto de bloque

- Un bloque de código determina la visibilidad de las variables declaradas dentro de él.
- Existen diferentes conceptos de bloque de código, como el más básico que se define usando las llaves.
- Pero además podemos crear un bloque de código a nivel de clase o dentro de un método usando también las llaves, que podemos marcar como **static** si fuera necesario (a nivel de clase).
- Podemos considerar un bloque de código una estructura **try/catch/finally**, donde cada pareja de llaves define una visibilidad propia.



netmind

WeKnowIT


## Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

## Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

[ ] netmind

WeKnowIT

# MI1717 – Parte 4

## Introducción a la Programación Orientada a Objetos en Java

## Herencia

# Programación Orientada a Objetos
# Herencia

## Módulo 4

- Modificadores de acceso
- Herencia simple y compuesta
- Concepto de abstracción: interfaces
- Clases abstractas y adaptadoras
- Polimorfismo: sobrescritura
- Trabajando con referencias de objetos
- Casting de referencias

# Modificadores de acceso

## Modificadores utilizados en las clases

- Podemos crear la declaración de una clase con cualquiera de los siguientes atributos:

| Modificador Java | Modificador .NET | Finalidad |
|---|---|---|
| **<default>** | **<default>** | Sin especificar |
| **public** | **public** | Accesible desde cualquier clase |
| **private** | **private** | Accesible desde la propia clase |
| **protected** | **protected** | Accesible desde la propia clase y desde clases que hereden de ella |
| **strictfp** | | Restringir cálculos de coma flotante para garantizar portabilidad |
| **static** | **static** | Variable única para todas las instancias |
| **final** | **final** | Variable de tipo constante |
| **abstract** | **abstract** | Requiere que la clase se herede para poder utilizarse |

- No es posible declarar una clase final y abstract a la vez. Por defecto si no declaramos un modificador de acceso, se establece como default/package (Java).
- Los packages (Java) y Assemblies (C#) delimitan el acceso.

# Modificadores de acceso (II)

## Uso de packages

- Un paquete o un assembly **es un contenedor para agrupar clases**, utilizando un espacio de nombres para evitar colisiones entre nombres.
- Los paquetes se almacenan **utilizando una estructura jerárquica** como en un espacio de nombres DNS o dominio.
- La utilización de un paquete (package) es opcional, aunque Java **sitúa las clases en un package denominado default**.
- Un paquete se declara en la primera línea de una clase, en minúsculas y cada dominio, separado por un punto:

```
package nombre.paquete;
```

# Modificadores de acceso (III)

## Uso de packages en Java

- Un paquete se importa después de la declaración del package (si existe) con:

**import nombre.paquete;**

- Al igual que en un DNS la estructura jerárquica de un paquete, se separa por puntos, cada uno de ellos corresponde con dominio o una carpeta:

**package pqt1.pqt2.pqt3;**

- Para importar una clase dentro de la estructura de un paquete, debe hacerse así:

**import java.awt.event.*;**

# Modificadores de acceso (IV)

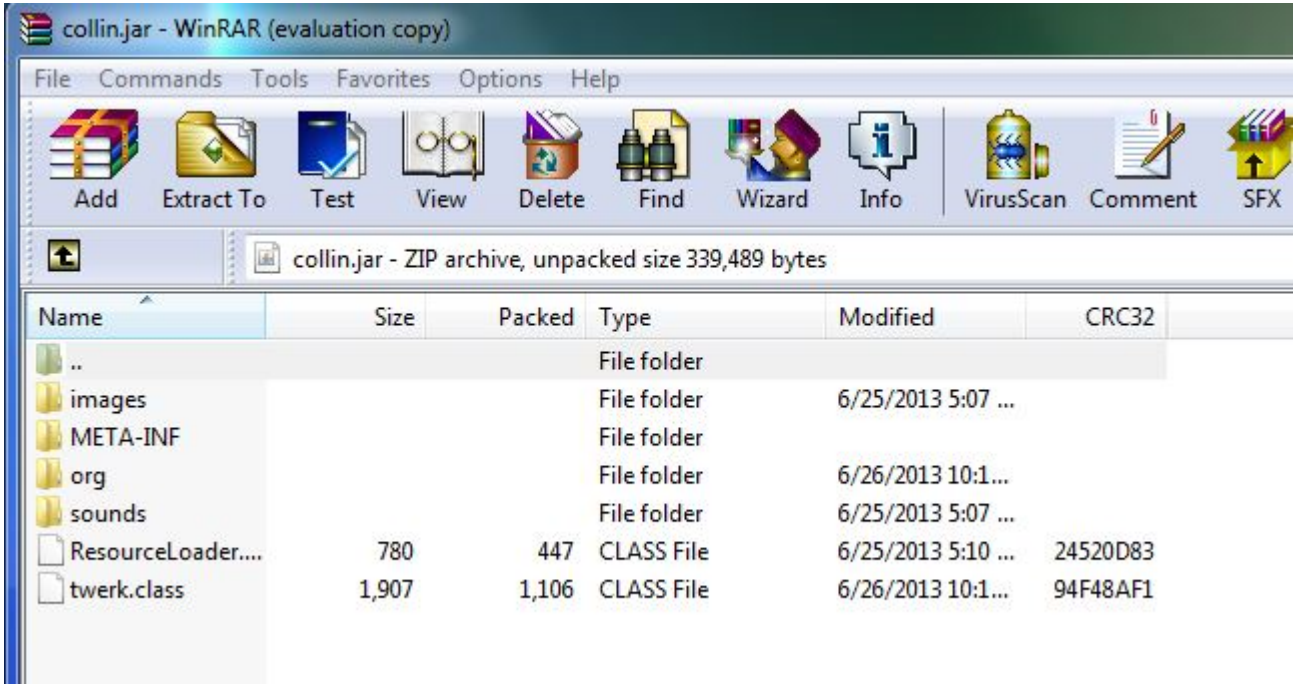
## Uso de packages en Java

- Java incorpora un número importante de packages conocido como las core API y que sirven para clasificar las clases por el tipo de tarea que realizan.

# Modificadores de acceso (y V)

## Archivos JAR (Java ARchive)

- Las clases con sus packages en Java se distribuyen en archivos JAR, estos archivos se crean usando la tecnología pkzip y se pueden manipular con una herramienta como Winzip, PowerArchive…



# Herencia simple y compuesta

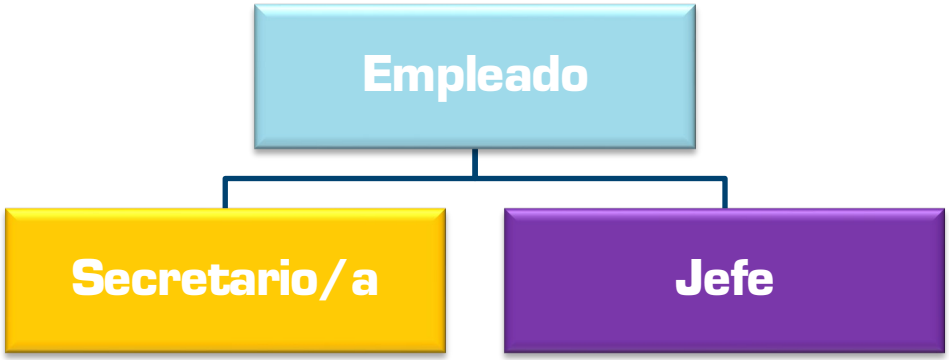
## Herencia en Java

- La herencia en Java es simple, es decir, únicamente una clase puede heredar de otra, aunque existe una herencia múltiple pero controlada: **interfaces**.
- En la herencia **el mecanismo de los constructores se invoca de forma automática**, siempre desde la superclase a las subclases.
- La cláusula **extends** (**inherits** en VB y **:** en C#) posibilita que una clase herede de otra, la clase de la que se hereda se conoce como base o SuperClass. Como Java no permite herencia múltiple, sólo es posible indicar una clase para heredar.
- No es posible heredar de una clase con el modificador **final. Las clases miembro, pueden ser también clases internas e interfaces.**

# Herencia simple y compuesta (II)

## Herencia en la POO

- La herencia es un mecanismo de los lenguajes de POO, que permite la reutilización y especialización del código. La herencia en Java es simple.
- Cuando una clase hereda o extiende de otra, incorpora todo aquello de la superclase o clase base, mientras que la que hereda se considera una subclase o clase derivada.



# Herencia simple y compuesta (III)

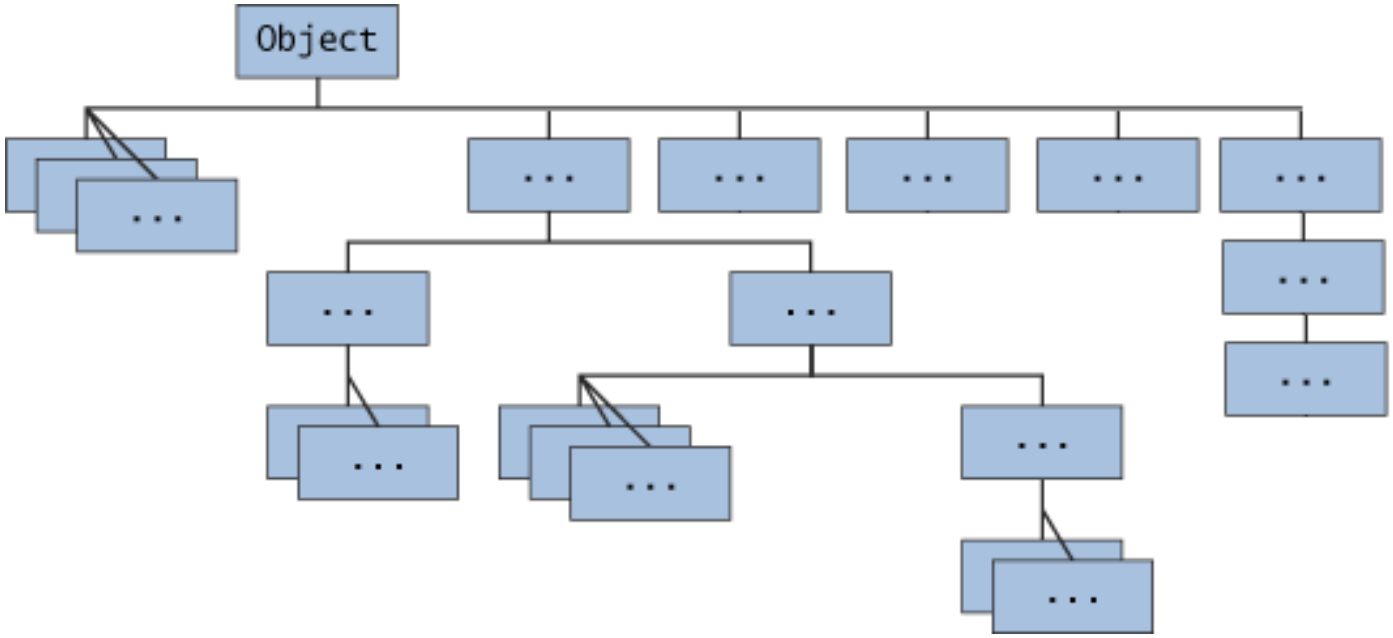
## Herencia en Java

- El mecanismo de **herencia produce un fuerte acoplamiento** entre las clases que lo utilizan, debemos usarlo con cuidado.
- **Favorece la reutilización de código** y consigue que este sea más fácil de mantener en el futuro.
- Podemos utilizar clases abstractas como superclases en la herencia, de esta forma podemos imponer unas reglas de sobrescritura.
- Es posible evitar que pueda heredarse de una clase, marcándola con el modificador **final**. Las variables o métodos marcados con **private** no podrán ser heredados.

# Herencia simple y compuesta (y IV)

## Creación de clases personalizadas

- **La clase más alta en la jerarquía de clases en Java y C# es Object**, absolutamente el resto de clases heredan de ella. Nosotros podemos crear nuestras propias clases añadiendo nuestro paquete a la jerarquía actual.



# Concepto de abstracción: interfaces

## Interfaces

- Tienen una gran parecido con las clases abstractas, sin embargo se diferencian en que:
    - **Se declara con el modificador interface**
    - **Todos sus métodos están sin implementar o son abstractos.**
    - **Las variables son implícitamente public static y final, y por lo tanto son constantes.**
- **Un interface es implícitamente abstracto**, pero no se utiliza el modificador abstract en su declaración.
- Es el mecanismo en Java que permite la herencia múltiple de forma controlada usando el identificador **implements o ":" en C#.** Una clase puede implementar múltiples interfaces.

# Clases abstractas y adaptadoras

## Clases abstractas

- Una clase abstracta es una clase que puede incluir métodos abstractos o sin implementación.
- **La clase abstracta se marca con la palabra abstract**, que también se utiliza para marcar los métodos abstractos.
- **Una clase que hereda de una clase abstracta está obligada a sobrescribir todos los métodos abstractos**. Sin embargo el resto de métodos son completamente opcionales  el sobrescribirlos.
- Las clases abstractas no pueden instanciarse y no poseen constructor. Una **clase adaptadora** es una clase abstracta que hereda de un interface y que no está obligada a sobre escribir todos sus métodos pero que lo hace para facilitar la sobre escritura.

# Polimorfismo: sobre escritura

## Polimorfismo con interfaces

- Una de las características fundamentales de la POO, es el polimorfismo, que no es otra cosa que la posibilidad de construir varios métodos con el mismo nombre, pero con comportamientos diferentes.
- **Esto nos permite crear código que se comporte de forma diferente en función de los datos que recibe en el mensaje.**
- Estos objetos recibirían el mismo mensaje global pero responderían a él de formas diferentes; por ejemplo, un mensaje "+" a un objeto ENTERO significaría suma, mientras que para un objeto STRING significaría concatenación ("juntar" strings uno seguido del otro).

# Trabajando con referencias de objetos

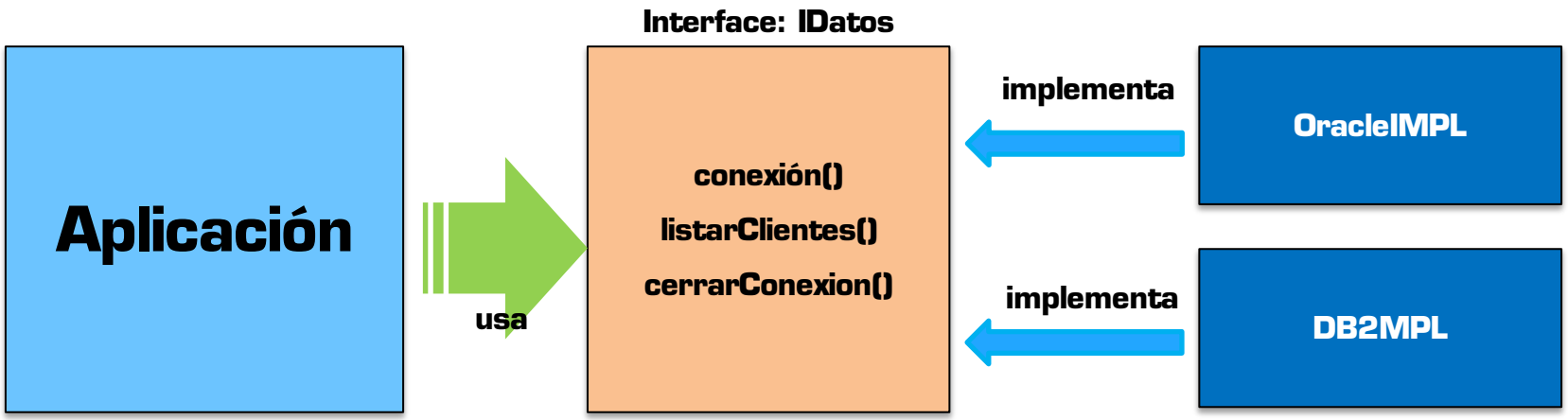
## Uso de interfaces como capa de abstracción

- Una de las facetas más interesante de los interfaces, es la de poder definir los límites de la aplicación y utilizarlos como capas de aislamiento de los riesgos de los cambios que puedan producirse en el futuro.
- Para ello se utiliza el concepto de referencias de objetos e implementaciones. Imaginemos una aplicación que usa inicialmente una BDD pero que en futuro puede cambiar, un interface puede darnos la solución al problema.
- Inicialmente creamos un interface donde se declaran las firmas de los métodos que se usarán para trabajar con la BDD y a continuación se crea una implementación, para esa BDD concreta.

# Trabajando con referencias de objetos (y II)

## Uso de interfaces como capa de abstracción

- En el futuro y cuando se produzca el cambio, creamos otra clase que implemente los mismos métodos pero con el código necesario para trabajar con la nueva BDD, las necesidades de refactoring se minimizan notablemente, gracias al uso de referencias de objetos:



# Casting de referencias

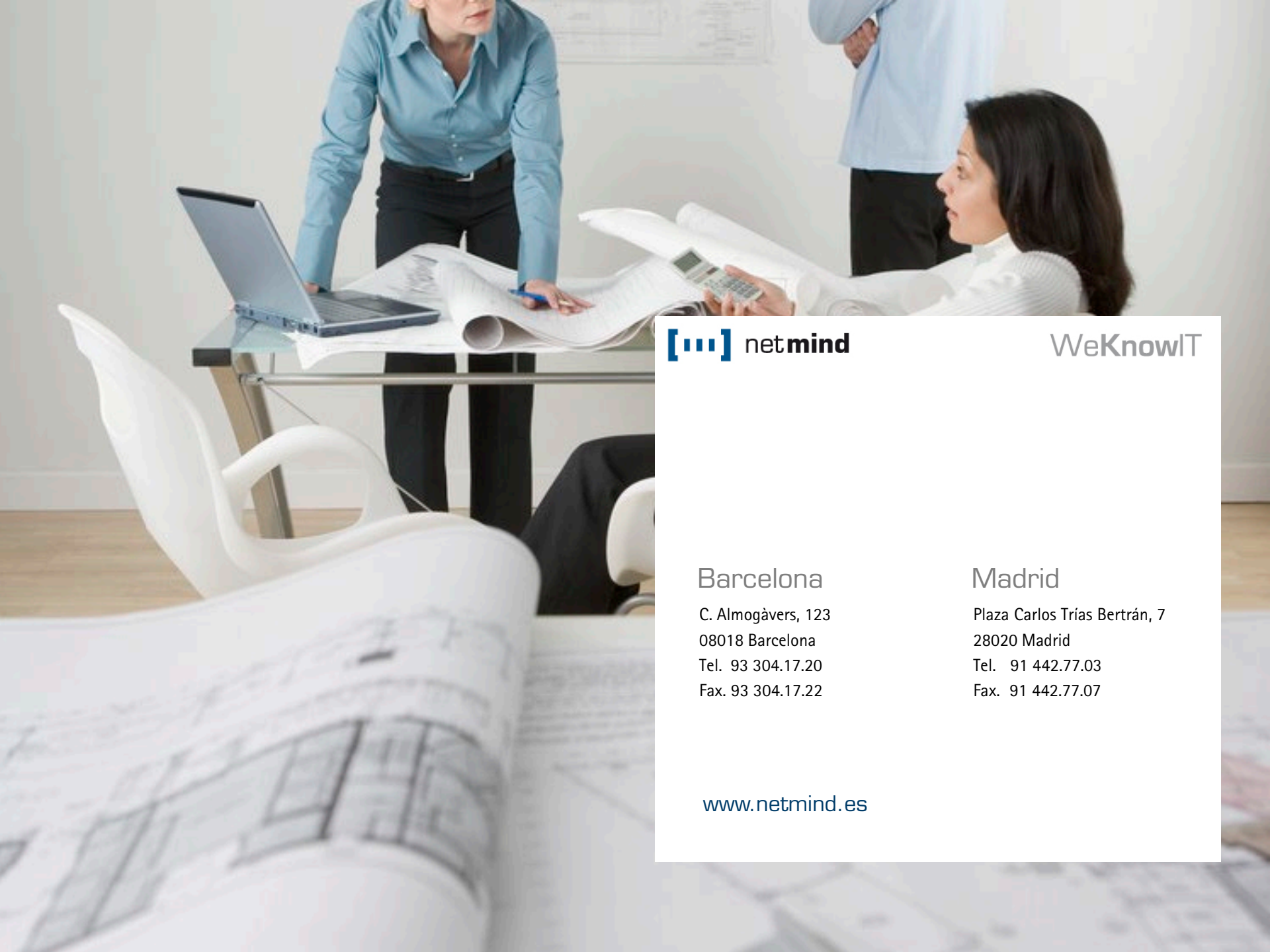
## Uso del casting con las referencias de objetos

- En el caso anterior se haría necesario utilizar el casting de referencias para poder crear el ejemplo:

**Aplicacion**

```
IDatos bd = new OracleIMPL();
//En el futuro comentamos la linea anterior y la sustituimos por:
//Idatos bd = new DB2IMPL(); //El resto no es necesario modificarlo.
bd.conexion()...
```

- De esta forma, podemos hacer un casting de las referencias OracleIMPL y DB2IMPL al decir que son de tipo IDatos (interface).
- La máquina virtual compara estas referencias en el momento de compilarlas.



netmind

WeKnowIT


## Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

## Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

netmind

WeKnowIT

# MI1717 – Parte 5

## Introducción a la Programación Orientada a Objetos en Java

## Estructuras de datos

# Programación Orientada a Objetos
# Estructuras de datos

## Módulo 5

- Estructuras condicionales
- Estructuras dinámicas List, Set y Map
- Patrón: Iterator
- Patrón: Observer
- Uso de Enumerator

# Estructuras condicionales

## Introducción

- Cualquier lenguaje de programación dispone de estructuras de control que permiten en determinados momentos el poder tomar decisiones en función de valores o situaciones imprevistas.
- Java y C# incorporan las siguientes estructuras de control y bucles:

| Estructura de decisión | Bucle |
|---|---|
| if / else if / else | while() |
| switch () | do while() |
| ternario ? : | for() |
| | for() como foreach |

# Estructuras condicionales (II)

## La estructura de decisión if / else if / else

- La estructura if realiza un evaluación booleana de un valor determinado, es decir, la expresión que escribamos entre los paréntesis debe devolver un valor true o false.
- Es opcional escribir el conjunto de llaves para delimitar el boque de código, si no las colocamos, sólo la primera línea a continuación del if participará del control del if.

Estructura de decisión

```
int var1 = 200;
if( var1 > 100 ){
        System.out.print("Es mayor que 100");
}else{
        System.out.print("Es menor que 100");
}
```

# Estructuras condicionales (III)

## La estructura de decisión switch

- La estructura de decisión switch se usa para cuando tenemos que evaluar una variable sobre un conjunto de valores conocidos o imprevistos. Hay que recordar sin embargo, que los tipos de datos con los que trabaja, son:

**byte – short – int – char**

Estructura de decisión switch

```
char c1 = 'a';
switch(c1){
        case 'a':
             System.out.print("var1 contiene a");
              break;
        case 'b:
             System.out.print("var1 contiene b");
              break;
        default:
             System.out.print("var1valor incorrecto de a");
              break;
}
```

# Estructuras condicionales (IV)

## La estructura de bucle while()

- La estructura while procesa el contenido de su bloque de código mientras se cumpla la condición booleana, ocasionalmente podemos escribir false para que se procese indefinidamente.
- Es opcional escribir el conjunto de llaves para delimitar el boque de código. El bucle do-while es idéntico pero se asegura que el bloque de código que contiene se procese al menos una vez.

Estructura de bucle while

```
int var1 = 200;
while( var1 > 100 ){
        System.out.print("Es mayor que 100");
        var1–;
}else{
        System.out.print("Es menor que 100");
}
```

# Estructuras condicionales (V)

## La estructura de bucle for

- La estructura for procesa el contenido de su bloque de código mientras se cumpla la condición booleana, ocasionalmente podemos escribir for(; ;) para que se procese indefinidamente.
- Es opcional escribir el conjunto de llaves para delimitar el boque de código. Esta formado por tres partes: inicialización, evaluación, incremento/decremento.

Estructura de bucle for

```
String s1 [ ]  = {“uno”, “dos”, “tres”};

for( int i = 0; i < s1.length; i++ ){
      System.out.print(“Valor:  ” + s1[ i ] );
}
```

# Estructuras condicionales (VI)

## La estructura de bucle for (foreach)

- La estructura **for llamada foreach** itera automáticamente sobre el contenido de un array o collection de objetos sin necesidad de controlar el valor de un índice..
- Dentro del bucle es necesario indicar el tipo de objeto que contiene la **collection/array** y declarar una variable para iterar sobre los elementos.

Estructura de bucle foreach

```
String s1 [ ]  = {“uno”, “dos”, “tres”};

for( String item : s1){
      System.out.print(“Valor:  ” +  item );
}
```

# Estructuras condicionales (VII)

## Uso de break

- Podemos usar break de tres formas:
    1. Para salir de una condición en un switch.
    2. Para salir de un bucle.
    3. Para usarlo con labels, como si fuera un goto.

Ejemplo

```
public static void main(String args[ ]) {
        for (int i = 0; i < 100; i++) {
        If ( i == 10) break; // termina el bucle cuando i vale 10
                System.out.println("i: " + i);
        }
        System.out.println(“Final del proceso");
}
```

# Estructuras condicionales (y VIII)

## Uso de continue y return

- Podemos usar continue en un bucle cuando deseemos que para un valor o conjunto de valores determinados el bucle realice unos saltos, es decir, deje de ejecutarse sin finalizar el bucle por completo. Por su parte return se usa para salir explícitamente de un método.

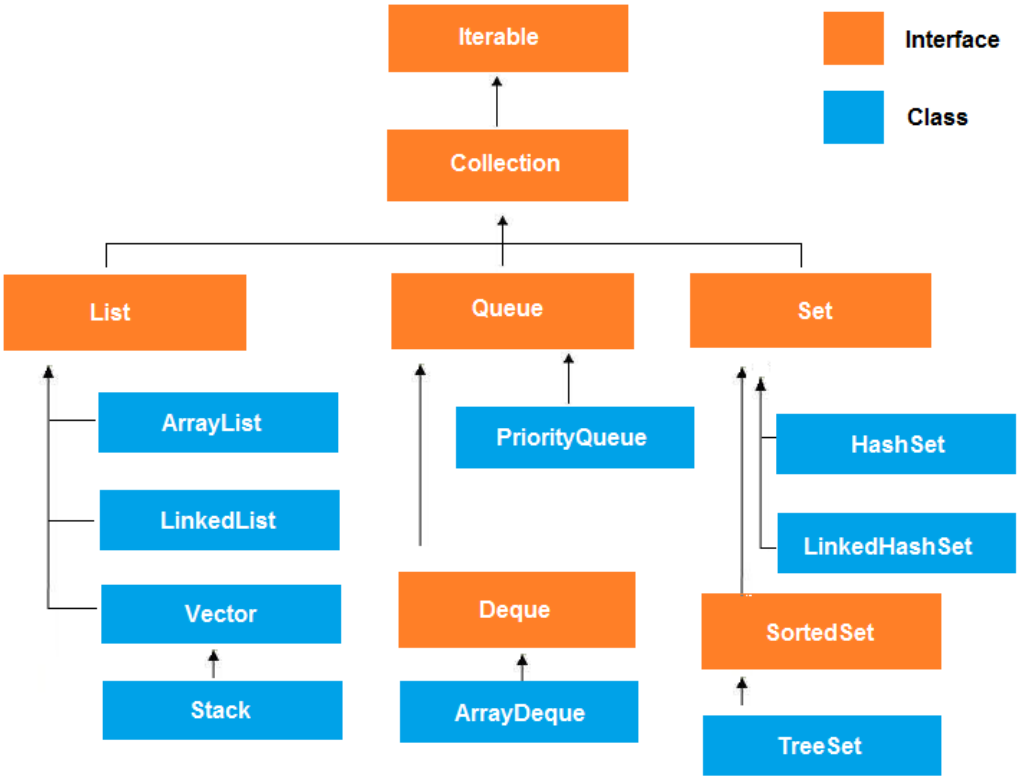
Ejemplo

```
public static void main(String args[ ]) {
        for (int i = 0; i < 100; i++) {
        if ( i == 10) break; // termina el bucle cuando i vale 10
                System.out.println("i: " + i);
                if (i%2 == 0) continue;
                System.out.println("salto");
        }
        System.out.println("Final del proceso");
}
```

# Estructuras dinámicas List, Set y Map

## El framework Collections

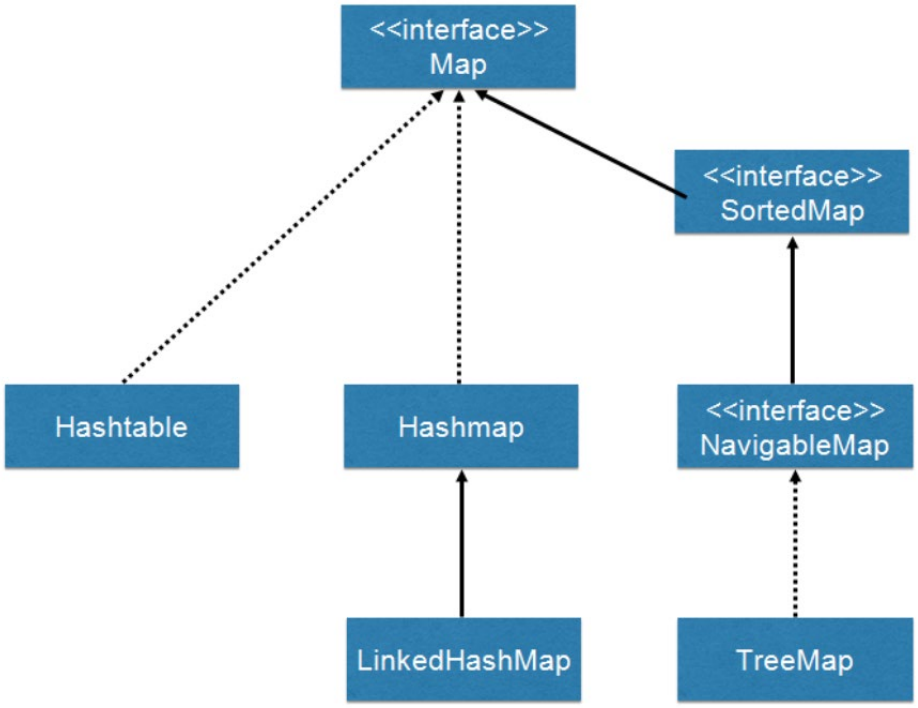
- Desde la versión 1.2 del JDK, Sun agrupó todas las estructuras de datos en el paquete java.util, con la intención de poder reutilizar funciones y crear una forma de acceso homogéneo a todas las implementaciones.



# Estructuras dinámicas List, Set y Map (II)

## El framework Collections

- Aunque las estructuras de tipo **Dictionary** o **Map** no implementan el interface **Collection**, pertenecen al mismo framework, en esta rama del árbol, se concentran las estructuras que almacenan la información en parejas de **clave=valor:**



# Estructuras dinámicas List, Set y Map

## El framework Collections y los genéricos

- Desde la versión 1.5 del JDK, Java incorpora un nuevo mecanismo que se llama genéricos.
- Los genéricos nos permiten realizar operaciones de boxing y unboxing de forma automática sin necesidad de usar los castings.

Ejemplo del uso de genéricos

```
public static void main (String args[ ])  {
        ArrayList <String>  lista = new ArrayList<String> ();
        lista.add("Barcelona");
        lista.add("Madrid");
        lista.add("Bilbao");
        Lista.add(new Integer(23));        //Error
}
```

# Estructuras dinámicas List, Set y Map

## El interface Comparable

- Es usado por el método **sort()** de Collections y Arrays. Para implementar este interface es necesario sobrescribir **compareTo()**.
- **Este método determina como serán ordenados los datos**, por ello, podemos implementar la lógica que deseemos para crear un mecanismo de ordenación personalizado.

Ejemplo del uso de Comparable

```
class DVDinfo implements Comparable<DVDinfo>  {
        public int compareTo(DVDinfo obj){
                return title.compareTo(obj.title());
        }
}
```

# Patrón: Iterator

## Propósito

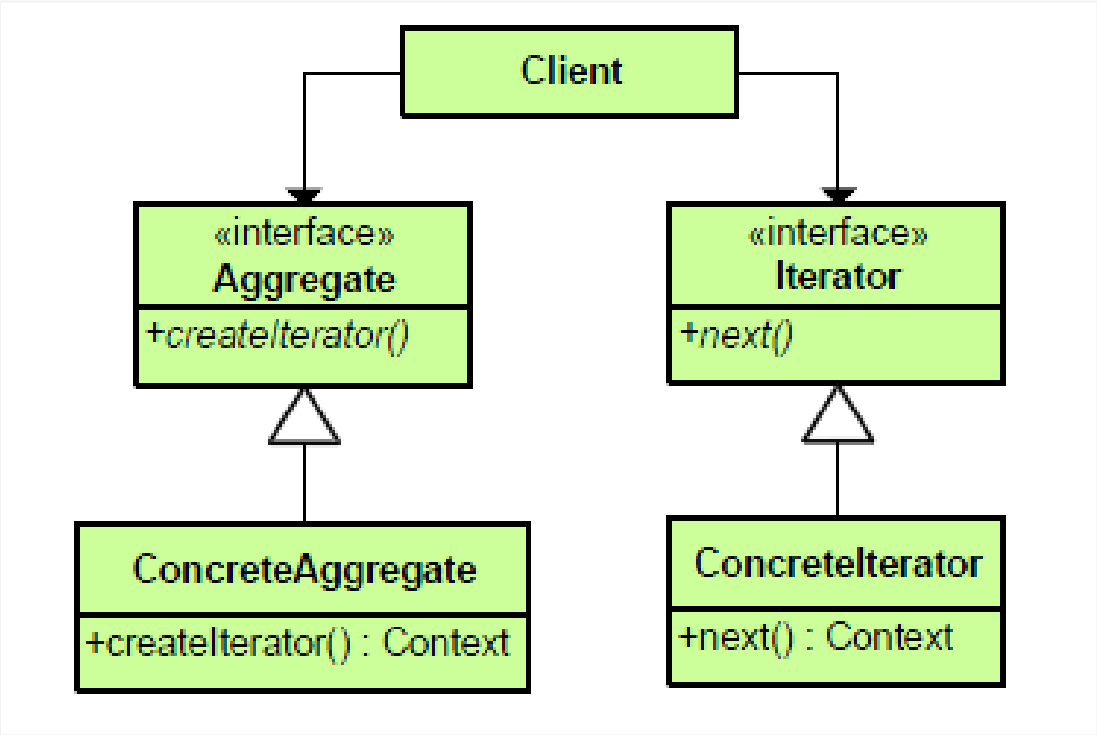
- Permite acceder a los elementos de una collection de objetos sin exponer su estructura, de esta forma podemos recorrer un objeto List o Set.

## Utilización

- Permite **acceder** a un **elemento**, **sin** necesidad de **acceder** al **objeto completo**.
- Se usa en condiciones de **acceso múltiple** o **concurrente** de objetos.
- Uso de un interface uniforme de acceso transversal, con pequeñas modificaciones en las implementaciones existentes.

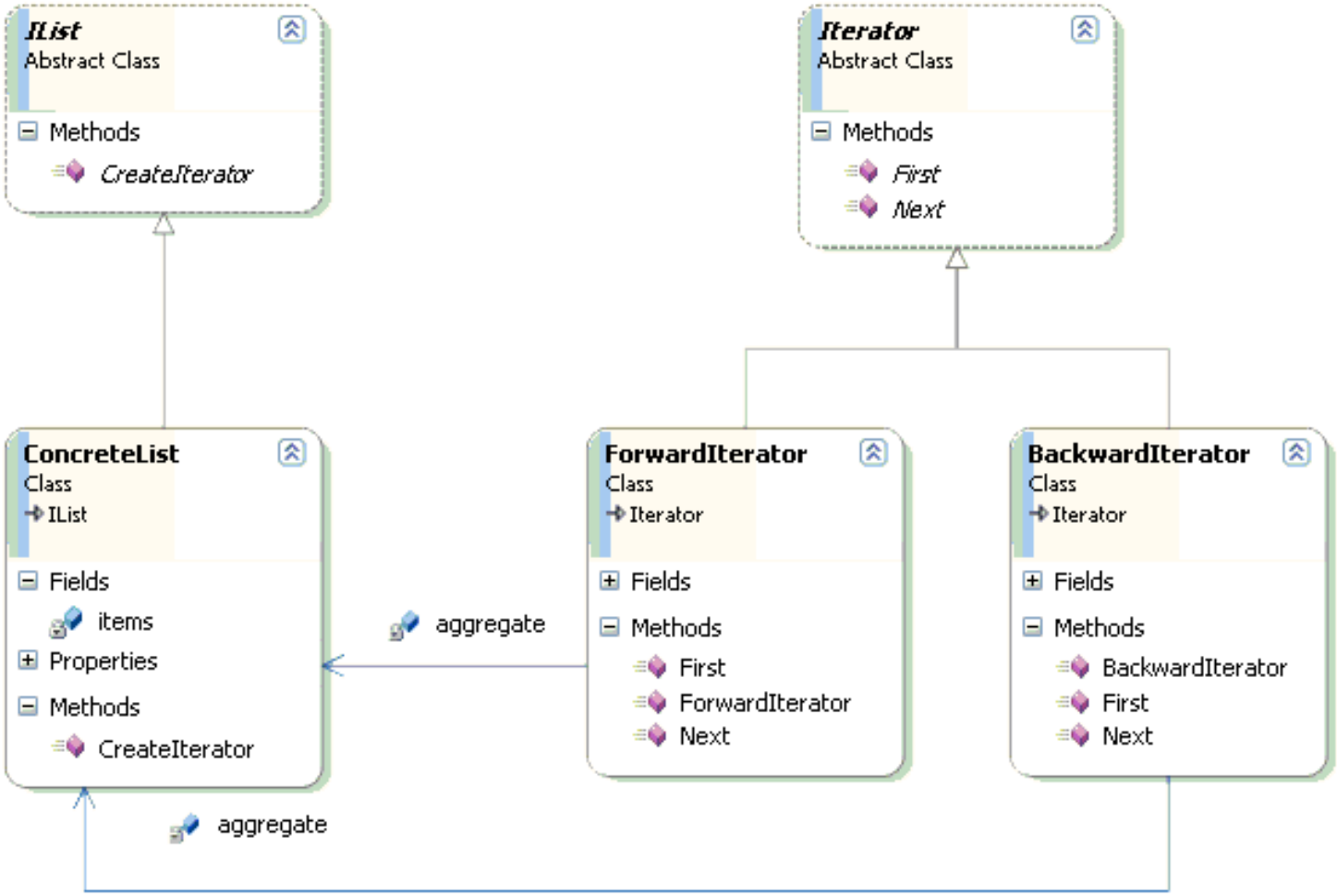
# Patrón: Iterator (II)

## Diagrama UML




- Existe una clase en Java con el mismo nombre que lo implementa, dentro del paquete “java.util”.

# Patrón: Iterator (y III)



# Patrón: Observer

## Propósito

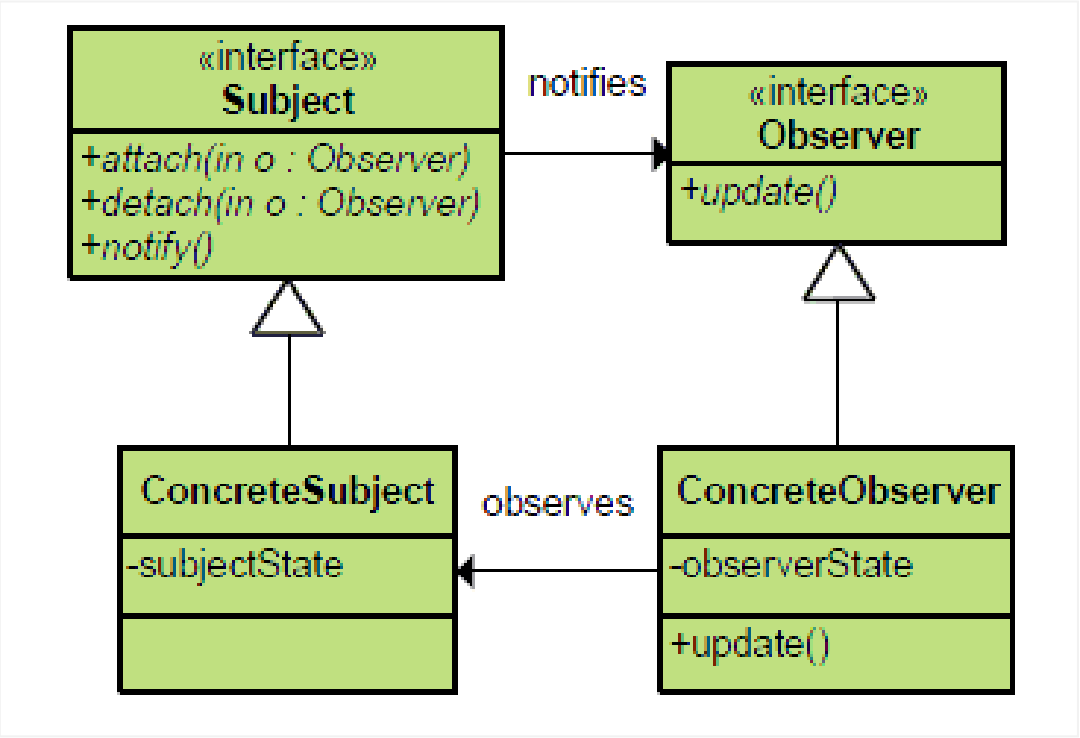
- Propagar eventos usando el modelo one-to-many, de tal forma que cuando un objeto modifica su estado, todos los observadores reciban un evento.

## Utilización

- En aquellos escenarios donde sea necesario, **propagar eventos cuando se produce un determinado tipo de suceso** de forma asíncrona.
- Es posible en tiempo de ejecución, **modificar el número y la actividad de los observadores.**
- Podemos **usar la implementación Observer-Observable** que viene incluida en el paquete **java.util**.

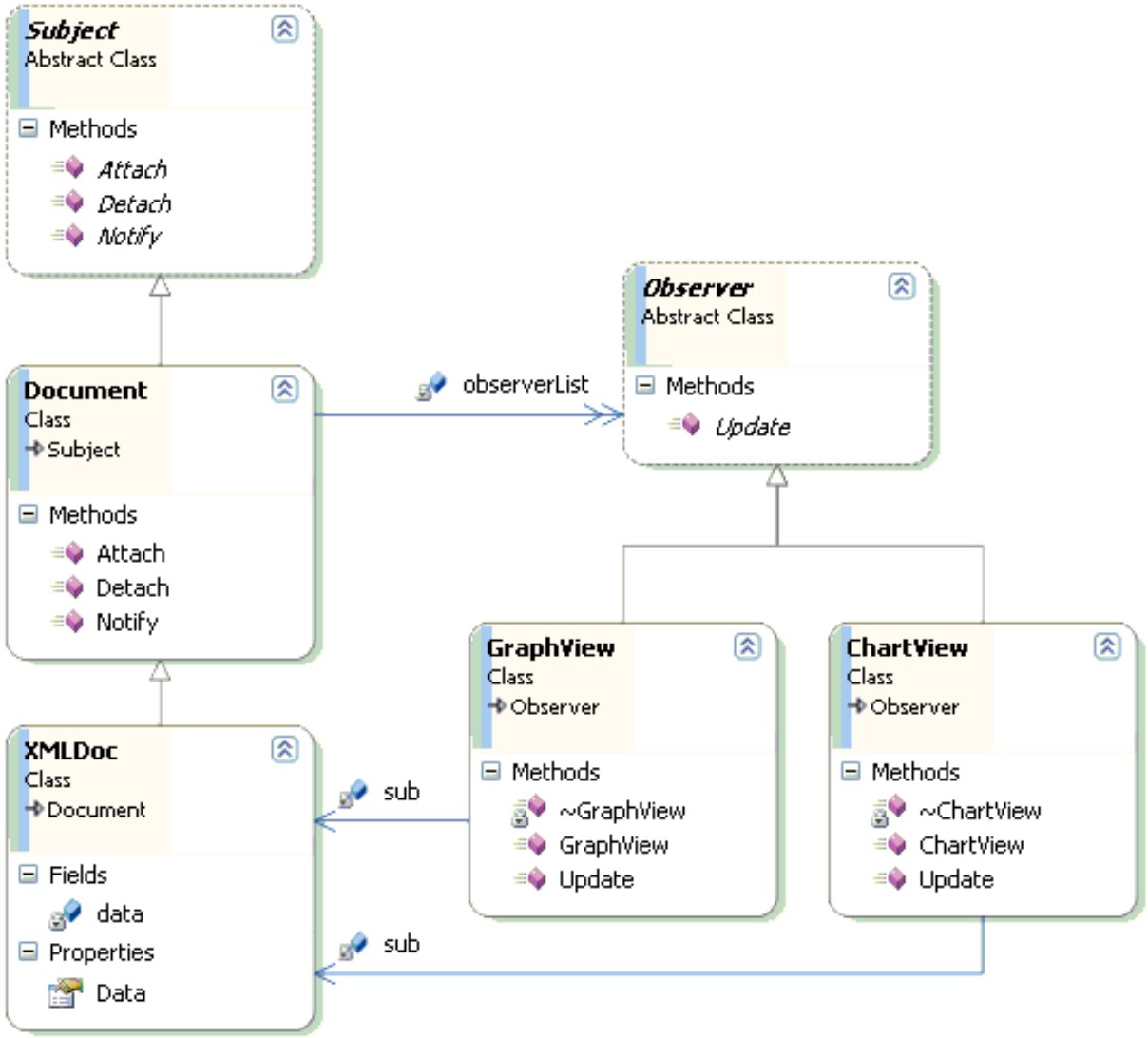
# Patrón: Observer (II)

## Diagrama UML




- Podemos **usar la implementación Observer-Observable** que viene incluida en el paquete **java.util**.

# Patrón: Observer (y III)



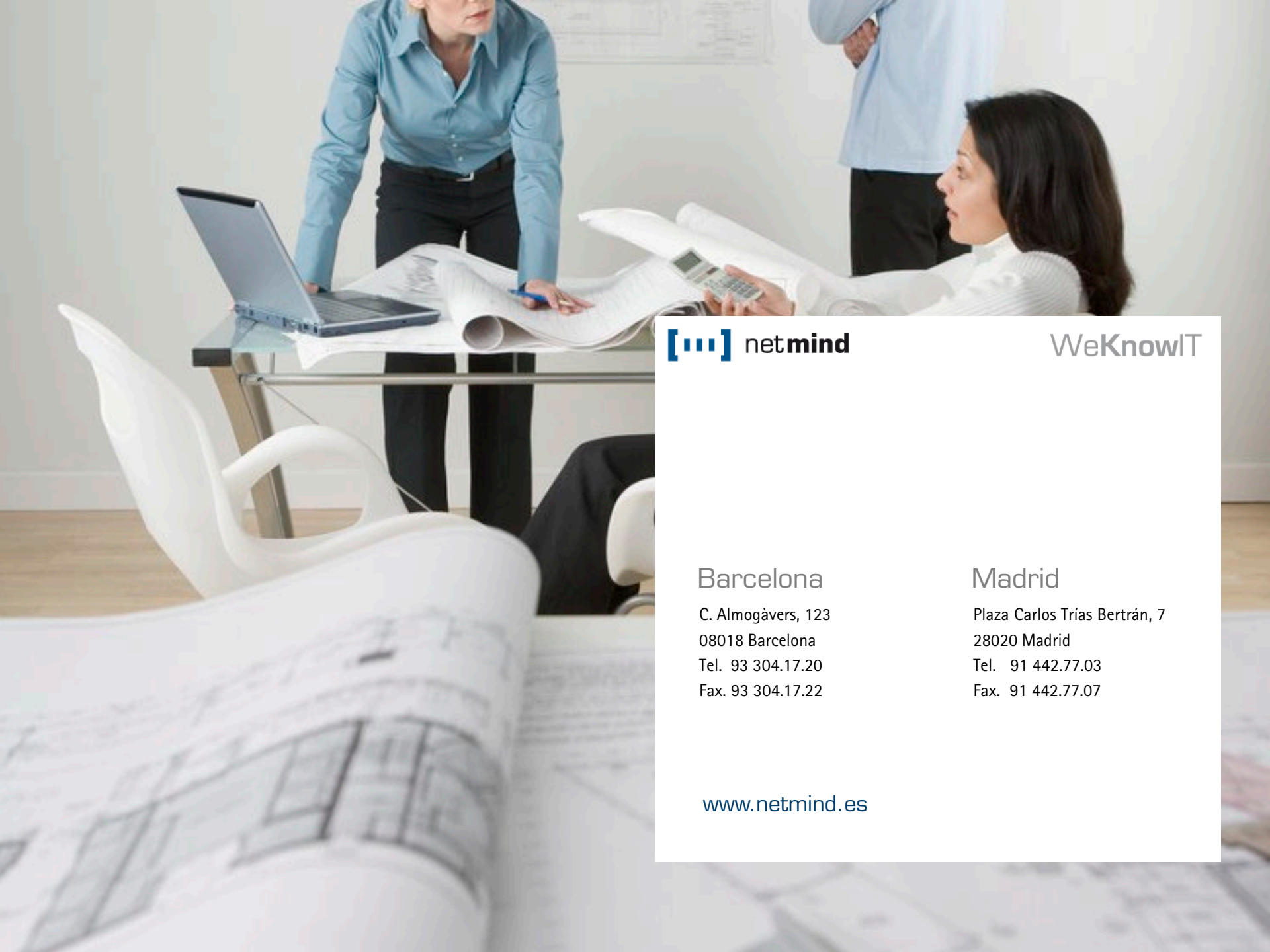
# Uso de enumerator

## El interface Enumeration

- El objeto que lo implementa genera un conjunto de elementos, uno cada vez, que permite recorrer una estructura de datos.
- La funcionalidad de este interface esta duplicada por la de Iterator, debemos seleccionar el más adecuado en cada momento.

Ejemplo del uso de Comparable

```
class Ejemplo  {
        public static void main(String[] args){
            for (Enumeration e = v.elements() ; e.hasMoreElements() ;) {
                    System.out.println(e.nextElement());
            }
        }
}
```



netmind

WeKnowIT


## Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

## Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

netmind

WeKnowIT

# MI1717 – Parte 6

## Introducción a la Programación Orientada a Objetos en Java

## Gestión de errores

# Programación Orientada a Objetos
# Gestión de errores

## Módulo 6

- Control de errores, excepciones en POO
- División de tareas
- Gestión de responsabilidades
- Jerarquía de excepciones
- Estructuras de captura

# Control de errores, excepciones en POO

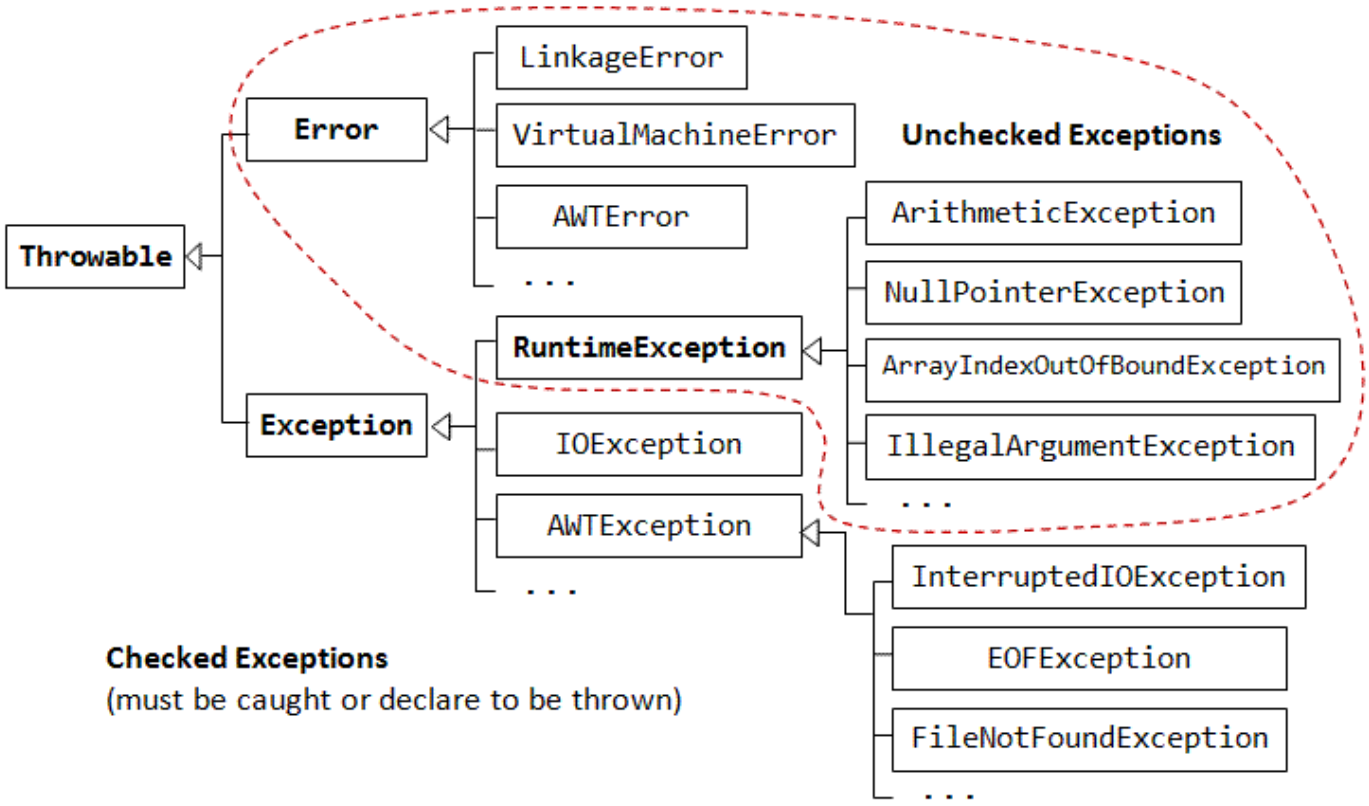
## Gestión de excepciones

- Condición excepcional ➔ se crea un objeto **Throwable** que se envía al método que la ha generado. Su gestión permite la detección y corrección de errores en ejecución.
- Simplifican los programas ➔ se diferencia el **código normal** del **código de tratamiento de errores.**
- Se crean programas mas robustos ya que en muchos casos si no se trata la excepción el programa no compila. **Sólo se deben usar cuando no se puede resolver la situación anómala directamente en ese contexto**.
- Es aconsejable declarar las excepciones específicas en las clausulas **catch**, ya que mejora el rendimiento de la JVM cuando se produce la excepción.

# División de tareas

## Jerarquía de excepciones

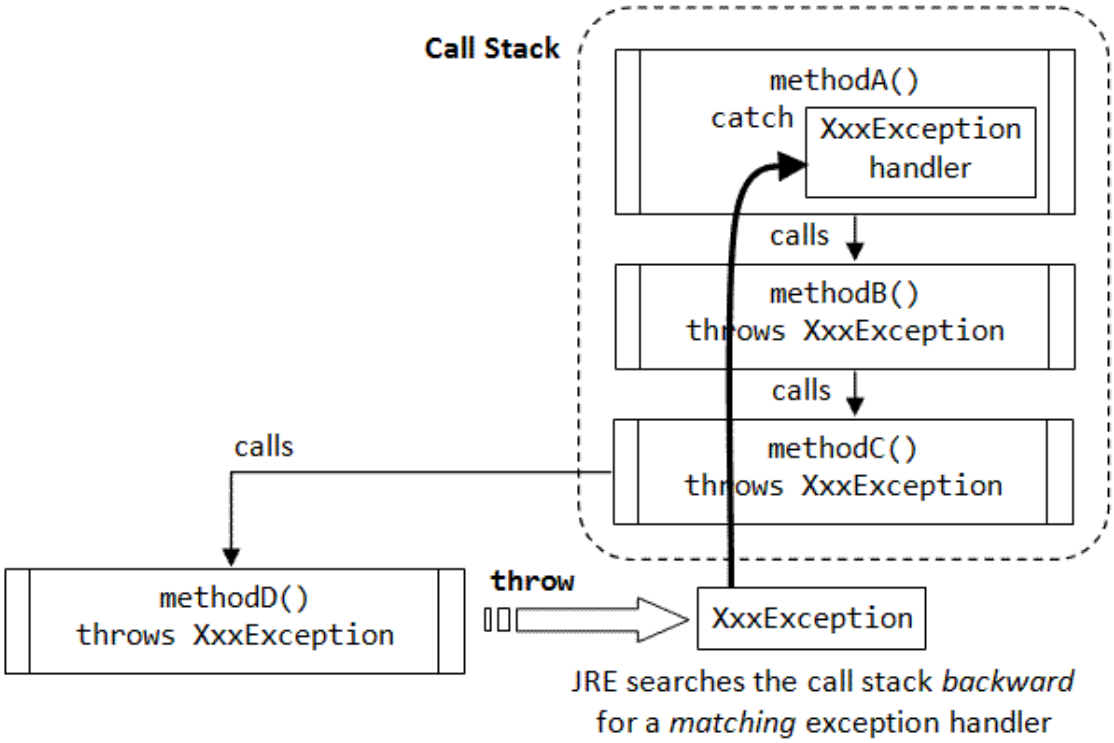
- Las excepciones de tipo Runtime, se consideran bugs del desarrollador y son su responsabilidad depurarlas y evitarlas. Existen **checked** y **unchecked** Exceptions.



# Gestión de responsabilidades

## Gestión de excepciones

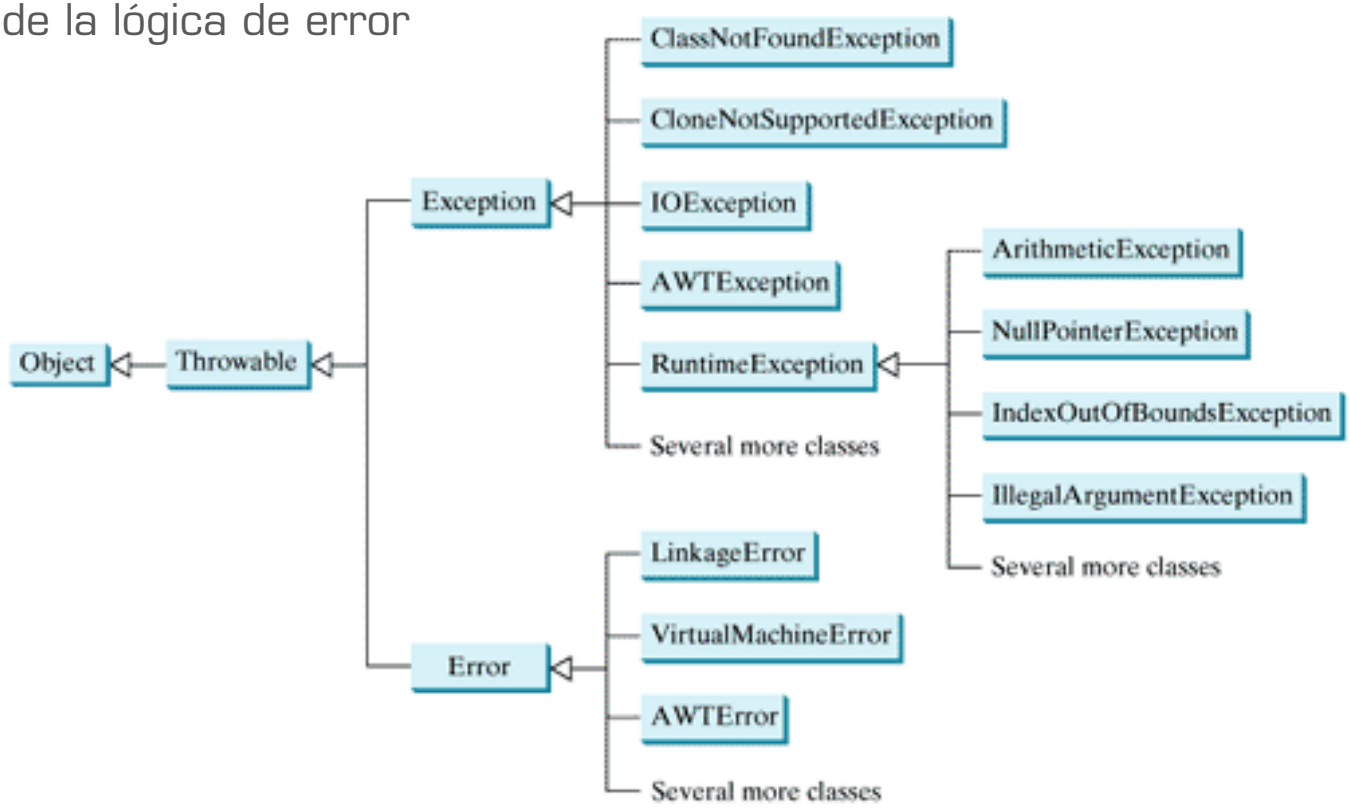
- Las excepciones son situaciones anómalas que aparecen durante la ejecución de un programa. En las aplicaciones desarrolladas con POO, la lógica de negocio se separa de la lógica de error



# Jerarquía de excepciones

## Algunas de las clases de Exception importantes

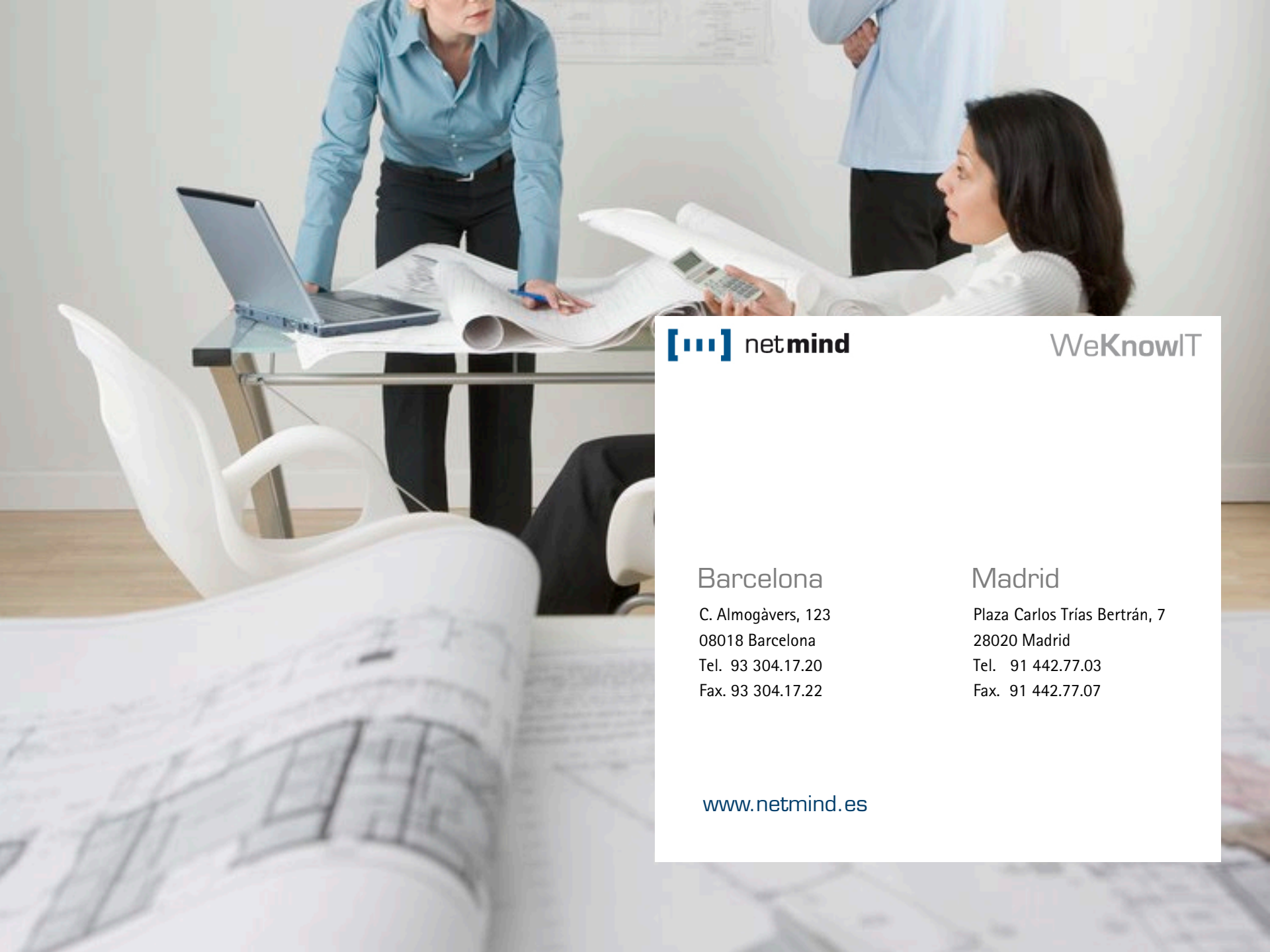
- Las excepciones son situaciones anómalas que aparecen durante la ejecución de un programa. En las aplicaciones desarrolladas con POO, la lógica de negocio se separa de la lógica de error



# Estructuras de captura

## Sintaxis en el uso de try / catch / finally

```
try {
   // bloque de código donde puede producirse una excepción
} catch (TipoExcepción1 e) {
   // gestor de excepciones para TipoExcepción1
   // se ejecuta si se produce una excepción de tipo TipoExcepción1
} catch (TipoExcepcion2 e) {
    // gestor de excepciones para TipoExcepción2
    throw(e); // se puede volver a lanzar la excepción – propagar
} finally {
   // bloque de código que se ejecuta siempre, haya o no excepción
}
```



netmind

WeKnowIT


## Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

## Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

netmind

WeKnowIT

# MI1717 – Parte 7

## Introducción a la Programación Orientada a Objetos en Java

## **Aplicaciones multithread**

# Programación Orientada a Objetos
# Aplicaciones multithread

## Módulo 7

- Características principales
- Sincronización de procesos
- La clase Thread
- Concurrencia organizada
- Evitar los deadlocks
- Garbage Collector

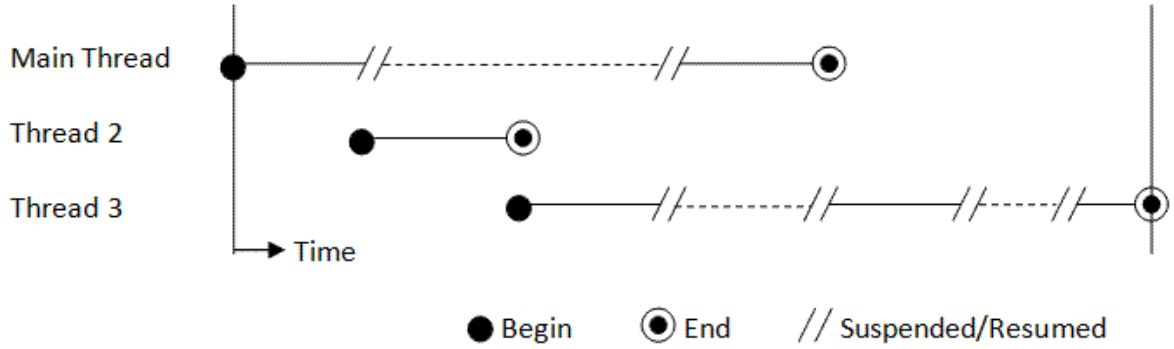
# Características principales

## Threads

- Crear un thread en Java es una tarea sencilla.
- Es un mecanismo que depende de la plataforma sobre la que se esté ejecutando la JVM.
- Un programa basado en threads puede realizar varias tareas de forma concurrente (Al menos ésa es la impresión que recibe el usuario, cuando el sistema cuanta con 1 sola CPU).
- Actualmente los sistemas operativos son multitarea, existen básicamente dos tipos:
    - La basada en procesos.
    - La basada en hilos de ejecución o threads.

# Características principales (II)

- Todas las aplicaciones disponen de varios threads ejecutándose durante el ciclo de vida de la aplicación como el thread main.




- Los sistemas operativos actuales son multitarea, de esta forma pueden realizar varias tareas de forma simultanea, de forma general existen dos tipos de multitarea:

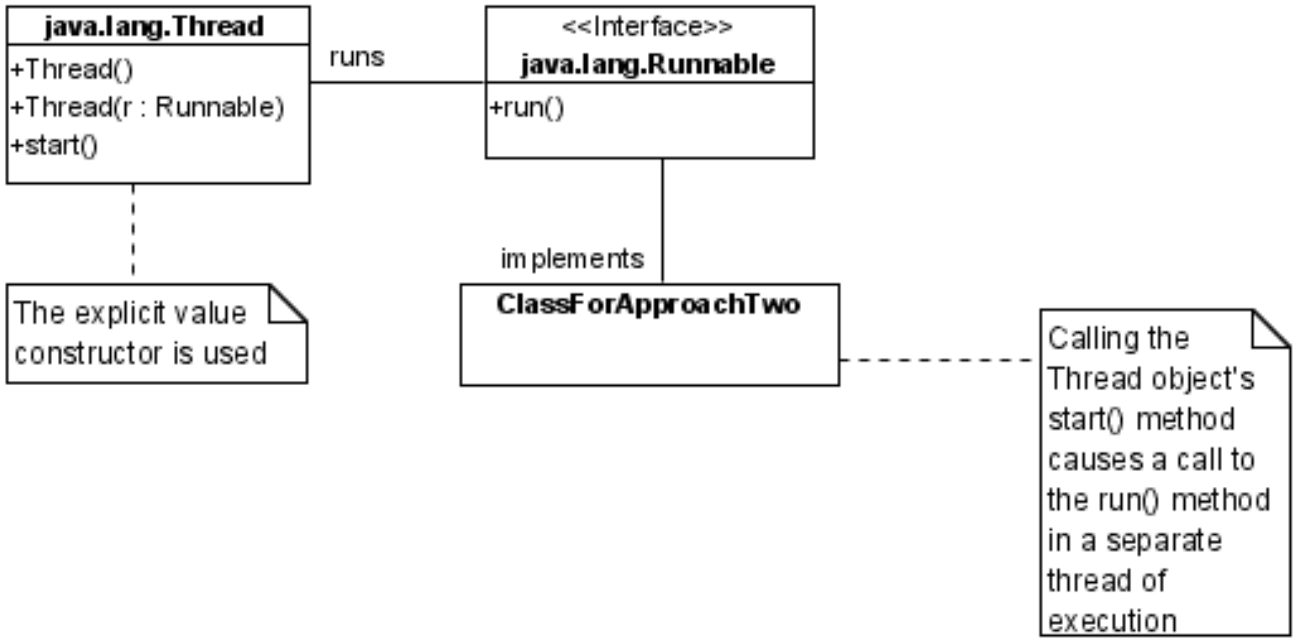
1. **Co-operative**, cada tarea cede voluntariamente el control de ejecución a otras tareas.
2. **Pre-emptive**, cada tarea dispone de un trozo (slice) de tiempo, finalizado el mismo, debe ceder el control a otra tarea.

# Características principales (III)

- **S.O. basado en procesos:**
  - Podemos considerar un proceso como un programa en ejecución, un ordenador puede ejecutar varios programas
  - Pensemos en aplicaciones como Word o Excel
  - En este tipo de multitarea, un programa es la unidad mínima.
  - Mientras la multitarea basada en procesos actúa sobre tareas generales la basada en hilos lo hace sobre los detalles.
- **S.O. basado en hilos de ejecución:**
  - **Hilo == tareas que un programa puede llevar a cabo**.
  - Pensemos en tareas como formatear un disco o imprimir un documento.
  - En este tipo de multitarea, un hilo de ejecución es la unidad mínima.
  - Un proceso puede tener varias tareas asociadas a él, ejecutándose.

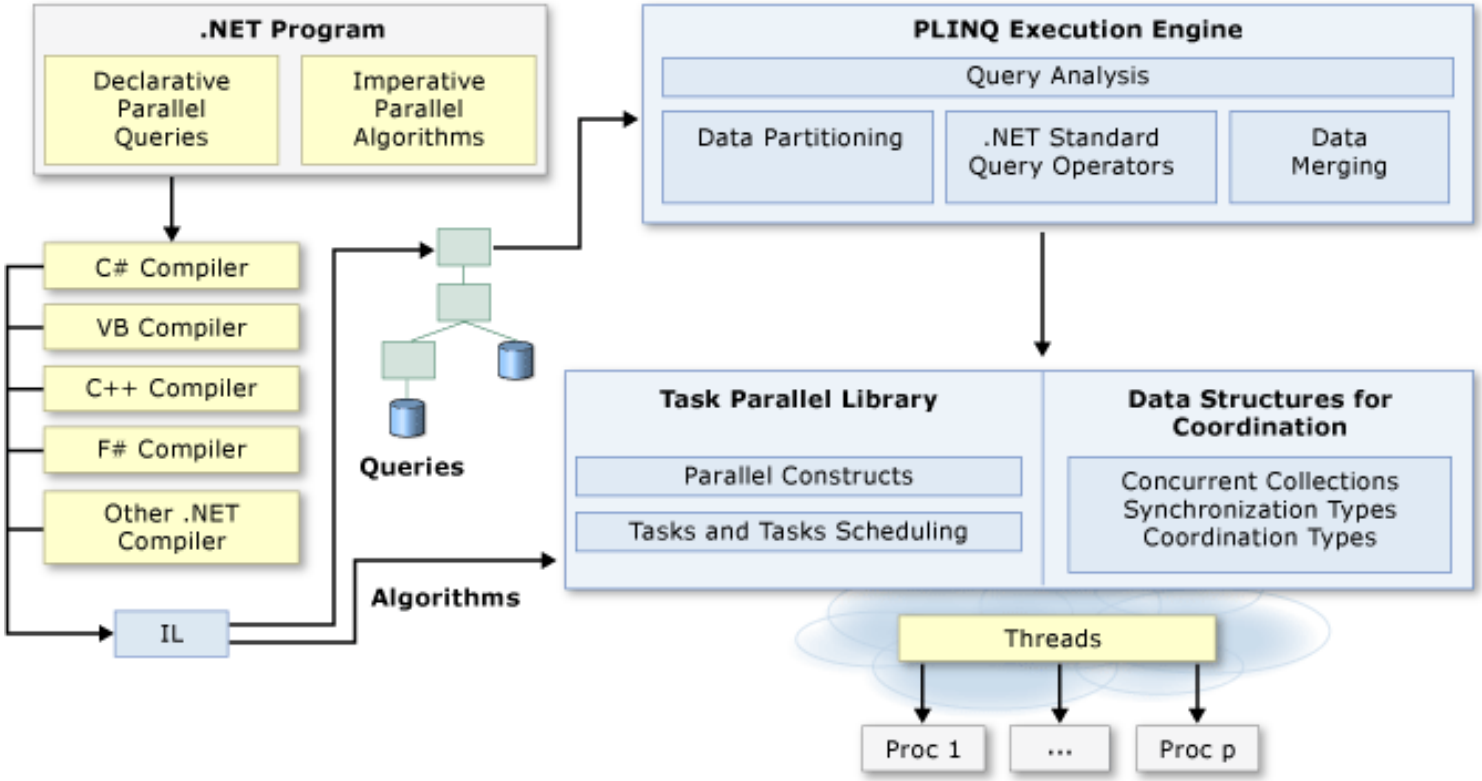
# Características principales (y IV)

- **Threads en Java**
  - Existen dos formas de trabajar con Threads en Java, extendiendo de la clase Thread o implementando el interface Runnable:



# Características principales (y IV)

- **Threads en .NET**
  - La forma de trabajar con Threads en .NET consiste en crear una instancia de la clase Thread y enviarla a un delegado ThreadStart. En las últimas versiones contamos con la Task Parallel Library:



# Sincronización de procesos

## La clase Thread

- Podemos crear threads a partir de la clase Thread y sobre escribiendo el método **run()**, que será invocado automáticamente en el momento que pase al estado de runnable. Podemos usar métodos como **join()** o **notify()** para sincronizar procesos:

| Método / Propiedad | Descripción |
|---|---|
| MAX_PRIORITY | Máxima prioridad que puede tener un hilo (thread) |
| MIN_PRIORITY | Mínima prioridad que puede tener un hilo (thread) |
| NORM_PRIORITY | Prioridad predeterminada que se asigna a un hilo (thread) |
| activeCount() | Devuelve el número de hilos activos en el grupo de hilos del hilo actual |
| currentThread() | Devuelve una referencia al objeto que representa al hilo en ejecución |
| getID() | Devuelve el identificador del hilo actual (thread) |
| getName() | Devuelve el nombre del hilo |
| getPriority() | Devuelve la prioridad del hilo |
| getThreadGroup() | Devuelve el grupo de hilos al que pertenece el hilo |
| getState() | Devuelve el estado del hilo actual |
| isAlive() | Comprueba si el hilo está vivo |

# La clase Thread

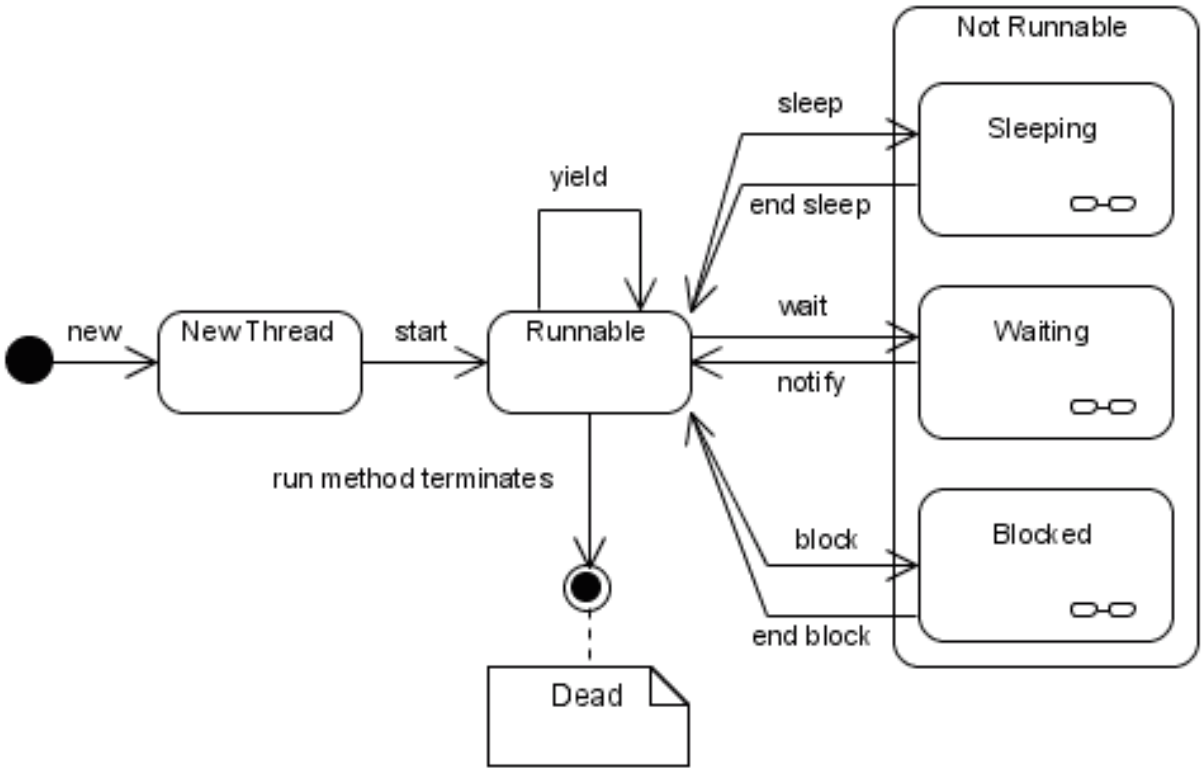
## La clase Thread

- El resto de métodos de la clase Thread se muestran a continuación:

| Método | Descripción |
|---|---|
| isDaemon() | Comprueba si el hilo es un hilo *daemon* |
| isInterrupted() | Comprueba si el hilo ha sido interrumpido |
| join() | Espera a que muera el hilo |
| run() | Si el hilo utiliza un objeto Runnable ➔ se ejecuta este método |
| setName() | Cambia el nombre del hilo |
| setPriority() | Cambia la prioridad del hilo |
| sleep() | Indica al hilo que se interrumpa durante los milisegundos indicados |
| start() | Indica al hilo que comience la ejecución, la JVM llama al método run de este hilo |
| yield() | Indica al hilo en ejecución que se pause temporalmente y permitir que entre otro hilo |
| setDaemon() | Marca el hilo como hilo *daemon* o *user* |

# Concurrencia organizada

- **Ciclo de vida de un Thread**
  - La JVM dispone de un mecanismo llamado schedule para trabajar con los threads sobre cualquier plataforma, la cual controla el ciclo de vida de un thread en Java:



# Concurrencia organizada

- Java proporciona una forma eficiente en la que varios threads pueden comunicarse.
- Esto reduce el tiempo de espera de la cpu, para ello existen varios métodos que permiten llevar a cabo esta tarea. Deben estar situados en un bloque sincronizado:

| Método | Descripción |
|---|---|
| **wait()** | Indica que el hilo que realiza la llamada espera hasta que que otros hilos comiencen a monitorizar utilizando los métodos **notify()** o **notifyAll()** |
| **notify()** | Despierta al primer hilo que llame al método **wait()** |
| **notyfyAll()** | Despierta a todos los hilos que llamaron al método **wait()**. El hilo con mayor prioridad se ejecutará primero. |

- Estos métodos deben estar encerrados en un bloque try/catch.
- Estos métodos trabajan con la agenda (schedule) de threads

# Evitar los deadlocks

- Para evitar que varios threads trabajen de forma simultanea sobre el mismo código o recurso, se hace necesario utilizar la sincronización de código en Java.
- Cada objeto en Java dispone de un bloqueo, podemos actuar sobre el para obtener un acceso exclusivo para un thread de forma concurrente.
- Cuando un thread entra en un bloque **synchronized** adquiere el bloqueo del objeto y no lo libera hasta haber finalizado su tarea:

Ejemplo

```
synchronized (object) {
        statement block;
}
```

# Garbage collector

- Ni en Java ni en .NET, no es necesario reservar zonas de memoria ni utilizar punteros para gestionar el mapa de memoria, por esta razón, tampoco existe la necesidad de liberar estos recursos.
- El recolector de basura lo hace por nosotros. El GC arranca cuando no existen referencias a un objeto y libera recursos.
- Básicamente lleva a cabo tres tareas:
    1. **Monitoriza objetos para conocer cuando dejan de usarse.**
    2. **Informa de los recursos que pueden liberar.**
    3. **Reclama estos recursos una vez destruido el objeto.**
- Trabaja en un Thread separado en el background para controlar a todos los objetos. Un objeto “existe”, mientras exista una referencia a el.

# Garbage collector (II)

- No existe la certeza de que el GC se ejecutará en los próximos 100 ms o en 2 segundos, ya que no es un proceso inmediato.
- Según la especificación de Java, la JVM y del CLR hará el mejor de los esfuerzos para iniciar el GC.
- Se puede reclamar el GC de una de las dos formas siguientes, aunque está desaconsejado:
    1. **Runtime.getRuntime().gc();**
    2. **System.gc() ;**
- Podemos asegurar que se realizará una acción, después de la destrucción de un objeto con el método finalize().
- Para ello basta con sobrescribir el método finalize() de esta manera:

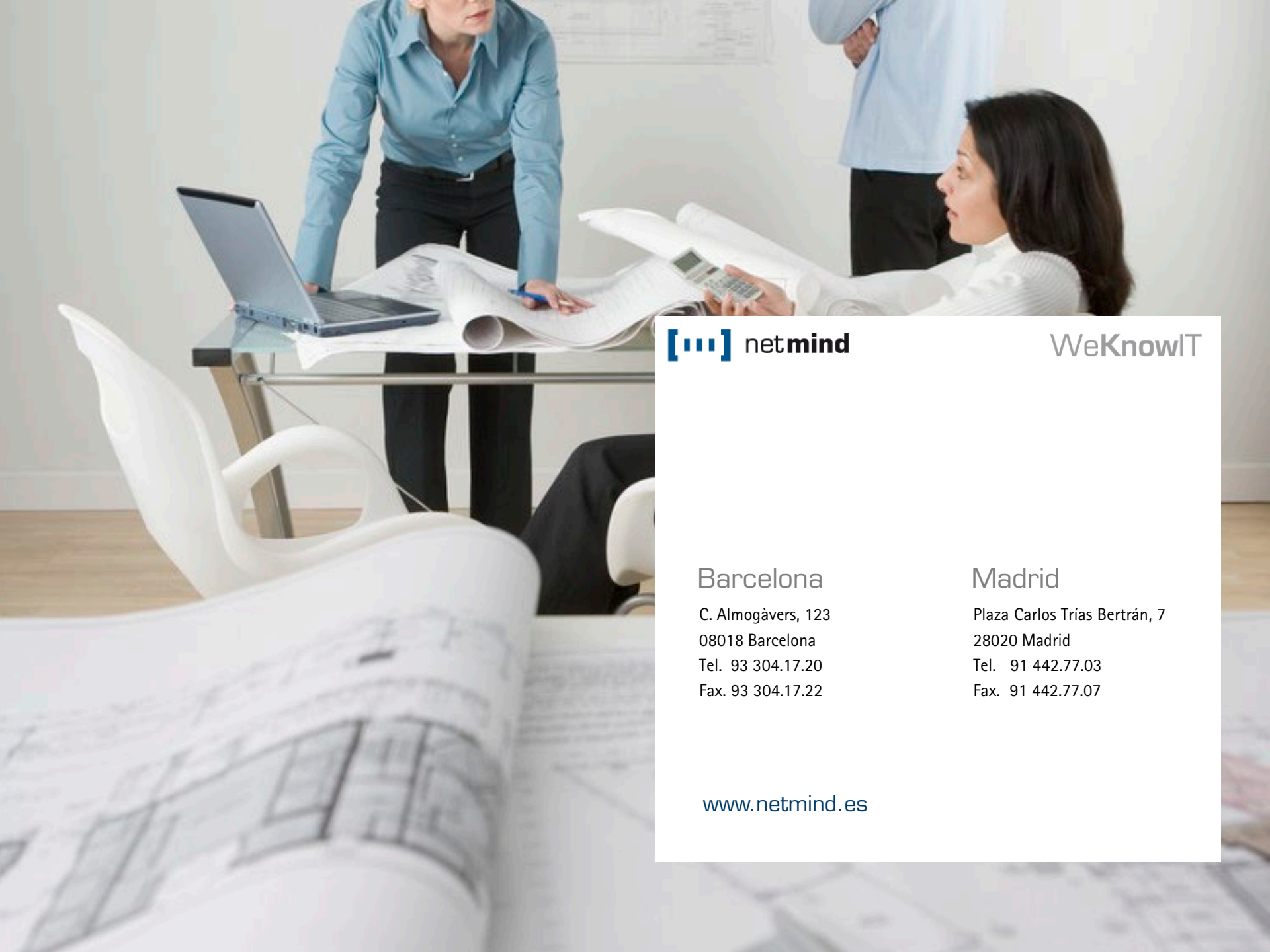
**protected void finalize() throws Throwable {}**

# Garbage collector (y III)

## Relación de candidatos a ser reciclados y su ciclo de vida

- Según su visibilidad y modificador, las referencias son recicladas en un determinado instante. Java y .NET usan un recolector generacional, que en función de donde se ha creado una referencia, le aplica un ciclo de vida diferente.

| Declaración | Duración |
|---|---|
| **Variable static** | Mientras se carga la clase |
| **Variable de instancia** | Durante la vida de la instancia |
| **Arrays** | Mientras exista una referencia al Array |
| **Parámetros de métodos** | Hasta la finalización del método |
| **Parámetros constructor** | Hasta la finalización del constructor |
| **Parámetros de Excepción** | Hasta la finalización de la claúsula catch |
| **Variables locales** | Hasta la finalización del bloque de código. En un bucle for, hasta su finalización. |



netmind

WeKnowIT


## Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

## Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

netmind

WeKnowIT

# MI1717 – Parte 8

## Introducción a la Programación Orientada a Objetos en Java

## Introducción al diseño de aplicaciones OO

# Programación Orientada a Objetos
# Introducción al diseño de aplicaciones OO

## Módulo 8

- Uso de capas: cliente / servidor
- Uso de capas: de 3 capas
- Uso de capas: de n capas
- Metodologías y el SDLC
- Ciclo de desarrollo de software
- Metodologías: RUP
- Metodologías: Agile manifestó
- Inyección de dependencias
- Best Practices

# Uso de capas: **cliente / servidor**

- Aplicaciones que desplegadas en el cliente, interactúan con una BDD en el servidor y al que pueden acceder de forma simultánea un número determinado de usuarios, se considera de 2-capas.
- Las características de este tipo de aplicaciones son las siguientes:
  - Clientes pesados, que pueden ser:
    - **Thin clients**, usan un ordenador conectado a la red sin disco duro.
    - **Thick clients o Fat clients**, utilizan la capacidad del cliente para procesar cierta tipo de lógica, p.e.: un applet.
- Conexiones dedicadas a BDD, lógica de negocio en BDD.
- Conexiones dedicadas con la BDD.
- Alta capacidad de administración, pero baja escalabilidad.
- Baja flexibilidad del sistema y portabilidad.

# Uso de capas: **de 3 capas**

- En este caso se añade un elemento más o capa en la arquitectura, y corresponde con el de la aplicación. De esta forma el cliente interactúa con una aplicación desplegada en el servidor la cual a su vez se comunica con la BDD.

- Las características de este tipo de aplicaciones son las que se describen a continuación:
  - **Reutilización de la lógica de negocio para diferentes clientes y sistemas.**
  - **Aumento de la escalabilidad.**
  - **Aumento de la flexibilidad del sistema.**
  - **Independencia de la base de datos**

# Uso de capas: **de n capas**

- En este caso se añaden más capas a la arquitectura, donde podemos tener múltiples capas, cada una de ellas especializada en un tipo de tarea concreta, como:
  - **Capa de presentación**
  - **Capa de negocio**
  - **Capa de persistencia, integración….**
- Las características de este tipo de aplicaciones son las que se describen a continuación:
  - **Bajo coste en la administración de los clientes**
  - **Alta flexibilidad**
  - **Capacidad en la tolerancia a fallos**
  - **Alta escalabilidad e independencia de la BDD**

# Metodologías y el SDLC

## Programación estructura vs POO

- La programación convencional o estructurada, se concentra en la lista de acciones secuenciales sobre un conjunto de datos, mientras que en la POO, las estructuras son el pivote de la programación.

"El término de Programación Orientada a Objetos
indica más una forma de diseño y
**una metodología de desarrollo**
de software que un lenguaje de programación."

- "La programación estructurada presta atención al conjunto de acciones que manipulan el flujo de datos (desde la situación inicial a la final), mientras que la programación orientada a objetos presta atención a la interrelación que existe entre los datos y las acciones a realizar con ellos."

# Ciclo de desarrollo de software

- **Se trata de un proceso lógico**, utilizado por una gran cantidad de empresas que sirve para afrontar el desarrollo de un proyecto de software.
- **Se ha dividido en diferentes fases**, cada una de ellas, trata de llevar a cabo una serie de tareas concretas y reducir la complejidad de desarrollar soluciones que se adapten a las necesidades del cliente.
- **Tiene un carácter secuencial**, y después de una fase se aborda otra, con un seguimiento lineal. Basa su éxito en que cada una de las fases se lleve a cabo con esmero y exactitud.
- Una **metodología** indica el procedimiento de trabajo para avanzar en la construcción del sistema. ***Cómo se realizan las actividades y con qué técnicas.***

# Metodologías: RUP

- **RUP** es la metodología que está asociada a la herramienta UML **Rationale Rose de IBM** y fue concebida para abordar proyectos de gran envergadura con un ciclo SDLC amplío.
- Es una metodología completa, de tipo iterativo, con un énfasis en una documentación muy precisa.
- Se fomenta la reutilización de componentes, existe training y tutoriales para todo el proceso
- Sin embargo, se trata de una metodología que no está orientada a proyectos de tamaño pequeño y mediano debido al volumen de información necesaria para trabajar.
- Los integrantes deben ser expertos en RUP, y el proceso en sí mismo, puede llegar a ser desorganizado.

# Metodologías: Agile manifesto

- Todas las Metodologías reunidas bajo Agile, comparten un conjunto de características que las hacen comunes, aunque cada una de ellas las implementa con su propio sello y están reunidas bajo este manifiesto:
  - Es imperativo la involucración de los usuarios.
  - El equipo debe tener el poder de tomar decisiones.
  - Los requisitos pueden cambiar, pero el timeline es fijo.
  - La captura de requerimientos debe ser de alto nivel: ligero y visual.
  - Desarrollar pequeñas versiones, incrementales e iterativas
  - Prever una entrega frecuente de versiones.
  - Una función debe estar terminada, antes de pasar a la siguiente.
  - Aplicar la regla 80/20.
  - El testeo debe estar integrado en todo el SLDC y debe ser frecuentes.
  - Todas las partes interesadas debe colaborar y cooperar activamente.

# Inyección de dependencia

- La inyección de dependencia, permite el efectivo desacoplamiento de las clases entre sí. Las dependencias directas proporcionan características colaterales a la aplicación, como:

    - **Fragilidad**, la introducción de cambios en la aplicación, produce comportamientos inesperados.
    - **Rigidez**, es difícil hacer cambios, afectan a muchas partes de la aplicación.
    - **Inamovilidad**, es difícil de llevar a cabo la reutilización de software, existe una escaza amortización del trabajo realizado.

# Inyección de dependencia (y II)

- La inyección de dependencia es uno de los mecanismos que permite fomentar el desacoplamiento entre las clases de una aplicación. El concepto es simple, en lugar de que la clase sea la encargada de buscar o estar asociada directamente a la clase o servicio que necesita de dos formas:

    1. **Liberándola de la necesidad de buscar el servicio**
    2. **Utilizando un interface para abstraerla del servicio**

- Actualmente la inyección de dependencia se utiliza usando las annotations, pero es posible usar un archivo XML para hacer las declaraciones.

# Best Practices

- La arquitectura de Java y otras plataformas que desarrollan con lenguajes orientados a objetos **comparten un conjunto de principios de arquitectura**, que si los seguimos, aportan unas características a nuestro diseño.

| Principio | Principio |
|---|---|
| **Encapsulación** | **Polimorfismo** |
| **Acoplamiento** | **Herencia con interfaces** |
| **Cohesión** | **Herencia de implementación** |
| **Composición** | |

- **Un equilibrio entre estos principios** puede dotar a las aplicaciones de ciertas **calidades sistémicas.**

# Best Practices (II) : Encapsulación

- La **encapsulación** es probablemente uno de los principios más utilizados y se usa **para ocultar la estructura interna de un objeto**, de esta manera podemos tratarlo como una caja negra que recibe mensajes y produce respuestas.
- Podemos gestionar la modificación de los valores de las variables.
- En Java se usa el concepto de JavaBean para denotar la encapsulación y tiene estas características:
  - La variables deben estar definidas con el modificador private.
  - Pueden tener métodos get/set/is.
  - Deben implementar el constructor sin argumentos (default constructor).
  - No es de carácter obligatorio, pero una best-practice es la de sobrescribir el método toString() de Object.

# Best Practices (III): Acoplamiento

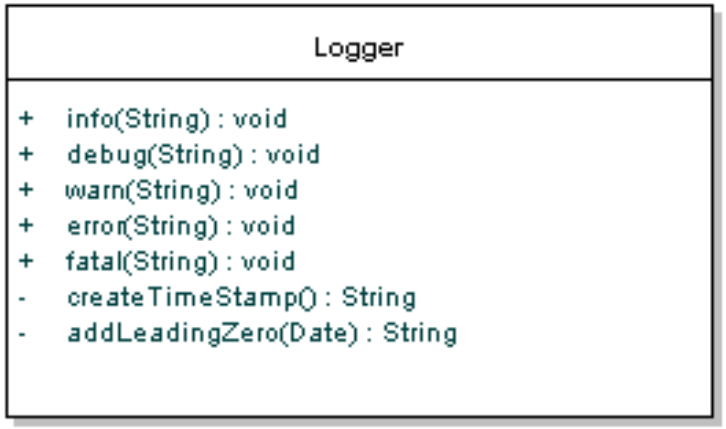
- El acoplamiento es una de las características clave de una aplicación en general. Definimos acoplamiento como el grado de dependencia que tiene una clase de la implementación de otra(s).
- ¿Pero, qué tiene de malo el acoplamiento?, pues que al crear una dependencia de una clase, cualquier modificación o cambio que hagamos en ella afecta a la primera y aporta una baja flexibilidad a la aplicación.

  Existen varios tipos de acoplamiento:

| Tipo de acoplamiento | Descripción |
|---|---|
| **Sin acoplamiento** | Dos clases que no tienen ningún tipo de relación. |
| **Bajo acoplamiento** | Una clase utiliza una instancia de otra. |
| **Bajo acoplamiento abstracto** | Una clase utiliza una instancia de otra a través de un interface. |
| **Alto acoplamiento** | Una clase hereda de otra clase. |

# Best Practices (IV): Cohesión

- La cohesión define el grado de coherencia de una clase, es decir, lo relacionados que están los métodos con la función que debe desempeñar una clase.
- Pensemos en una clase Logger como la de la imagen, ¿Qué nos puede resultar extraño, o mejor dicho, le hemos otorgado alguna responsabilidad extra?:

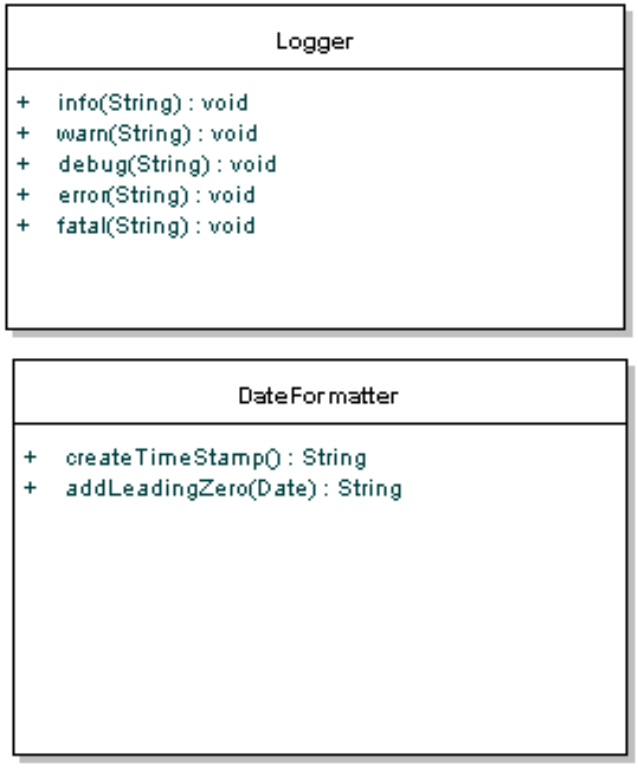



**Baja cohesión**

- El método createTimeStamp() y addLeadingZero(), no deberían ser una responsabilidad de esta clase.

# Best Practices (V): Cohesión (y II)

- Una solución que permite la reutilización de código en el ejemplo anterior y crea un tipo de cohesión elevado, es la siguiente:
- Hemos decidido separar estos dos métodos para conseguir un objeto de Logger, justo con las responsabilidades que debe tener.
- Los dos métodos, los hemos colocado en otra clase, que nos permitirá reutilizarlos y que fomentará un bajo acoplamiento.
- Un bajo acoplamiento a su vez dota de flexibilidad a una aplicación.




**Alta cohesión**

# Best Practices (VI): Composición

- En lugar de usar la herencia como único mecanismo para conseguir clases más especializadas, podemos usar el concepto de composición en su lugar, también llamado reutilización de caja negra.
- La composición nos permite reutilizar funcionalidad, creando objetos complejos a partir de objetos básicos.
- El objeto contenedor es responsable del ciclo de vida del objeto contenido. A su vez si el objeto contendor es reciclado, el objeto contenido desaparece igualmente.

```
public class Ordenador {
        private DiscoDuro hdd1;

        public Ordenador() {
        }...
}
```

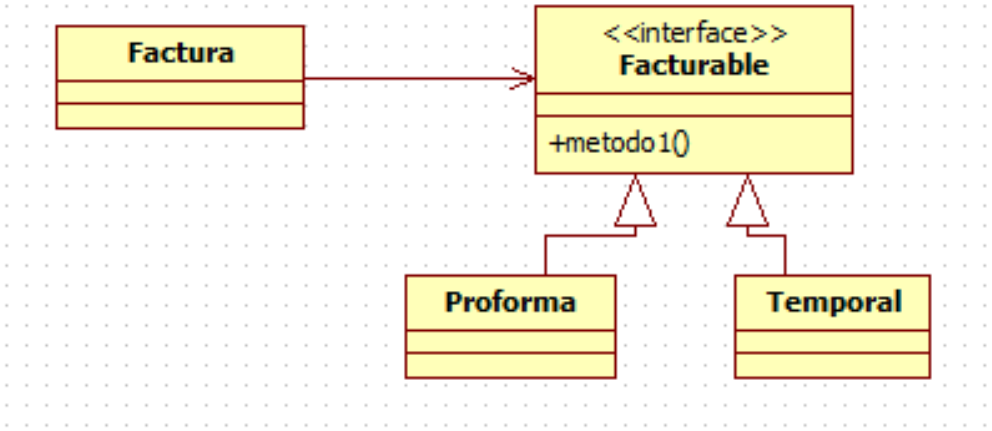
# Best Practices (VII): Polimorfismo

- El polimorfismo nos permite por ejemplo, disponer de varios métodos con el mismo nombre, pero distinto número o tipos de argumentos, que al ser invocados en runtime, la JVM decidirá cual es el más representativo o se ajusta más a las características de la llamada.
- Podemos asignar el valor de una variable a diferentes tipos en periodo de ejecución.
- Nos permite crear código genérico y que no dependa de una subclase específica.

```
public static void main(String args) {
        calcular(12.5, 8.8);
}
public static calcular(int c1, int c2)   {…}
Public static calcular(double c1, double c2) {…}
```

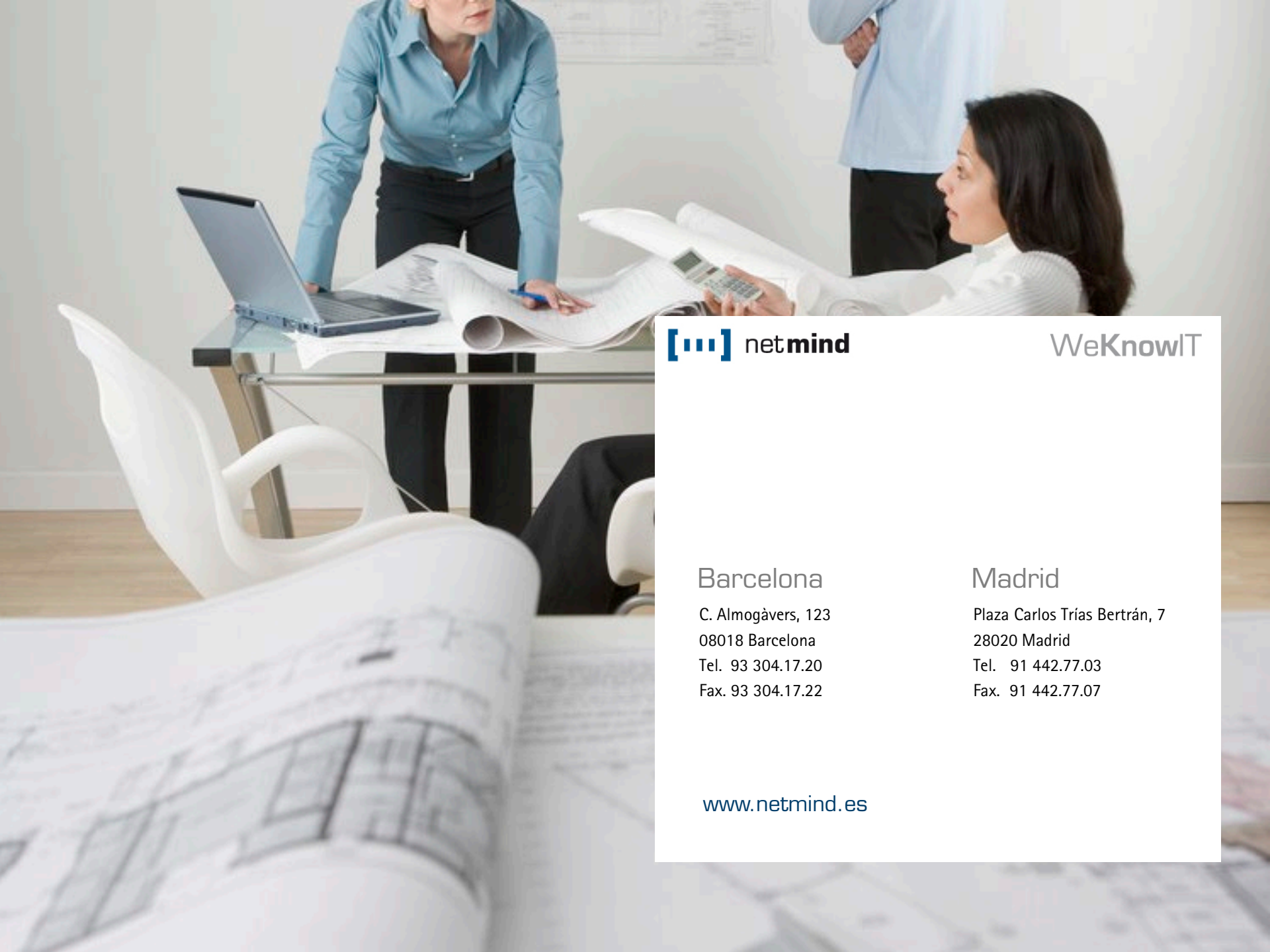
# Best Practices (VIII): Herencia con interfaces

- Utilizando la herencia a través de interfaces, podemos separar el interface de la implementación, desarrollando el principio de desarrolla para el cambio.
- Podemos usar la metáfora de un coche, donde disponemos de un acelerador, un embrague y un freno que nos abstraen de la implementación del motor que cada uno incorpora.
- Podemos crear una clase extensible sin necesidad de que sea necesario modificarla.



# Best Practices (IX): Herencia de implementación

- Es el **tipo de acoplamiento más alto**, usado con buen criterio puede ser un excelente mecanismo de desarrollado, pero si se abusa de el, crea clases excesivamente acopladas.
- Este fuerte acoplamiento ➔ una alta dependencia de otras clases y penaliza la flexibilidad general de la aplicación.
- Este tipo de herencia se construye con la palabra **extends** ( **:** en C#)**,** y permite que una clase herede los atributos y métodos que hayan sido definidos para ello.
- Permite eliminar el código redundante y repetitivo, situándolo en las subclases.
- Permite crear grupos de clases en estructuras jerárquicas.



netmind

WeKnowIT


## Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

## Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

netmind

WeKnowIT

# MI1717 – Parte 9

## Introducción a la Programación Orientada a Objetos en Java

## UML en la POO

# Programación Orientada a Objetos
# UML en la POO

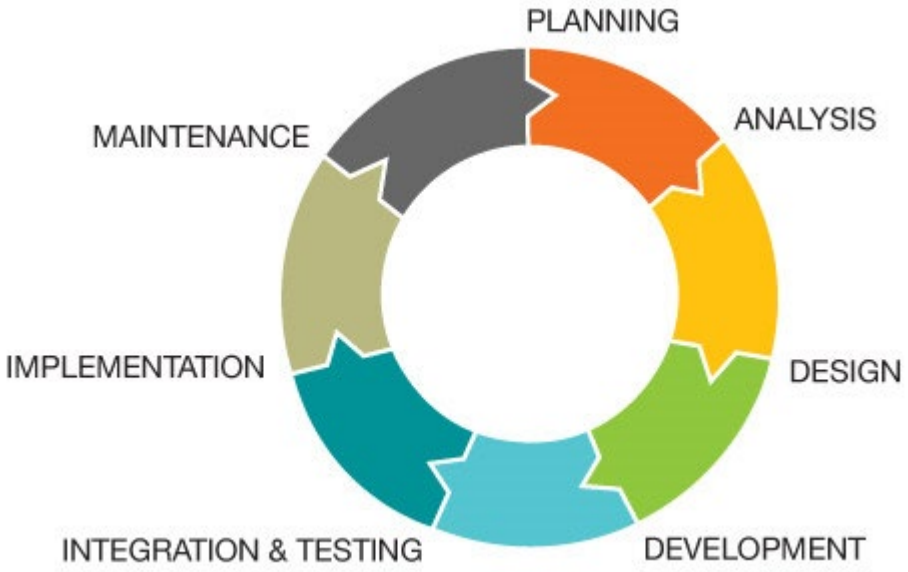
## Módulo 9

- Nociones de UML
- Modelando la aplicación

# Nociones de UML

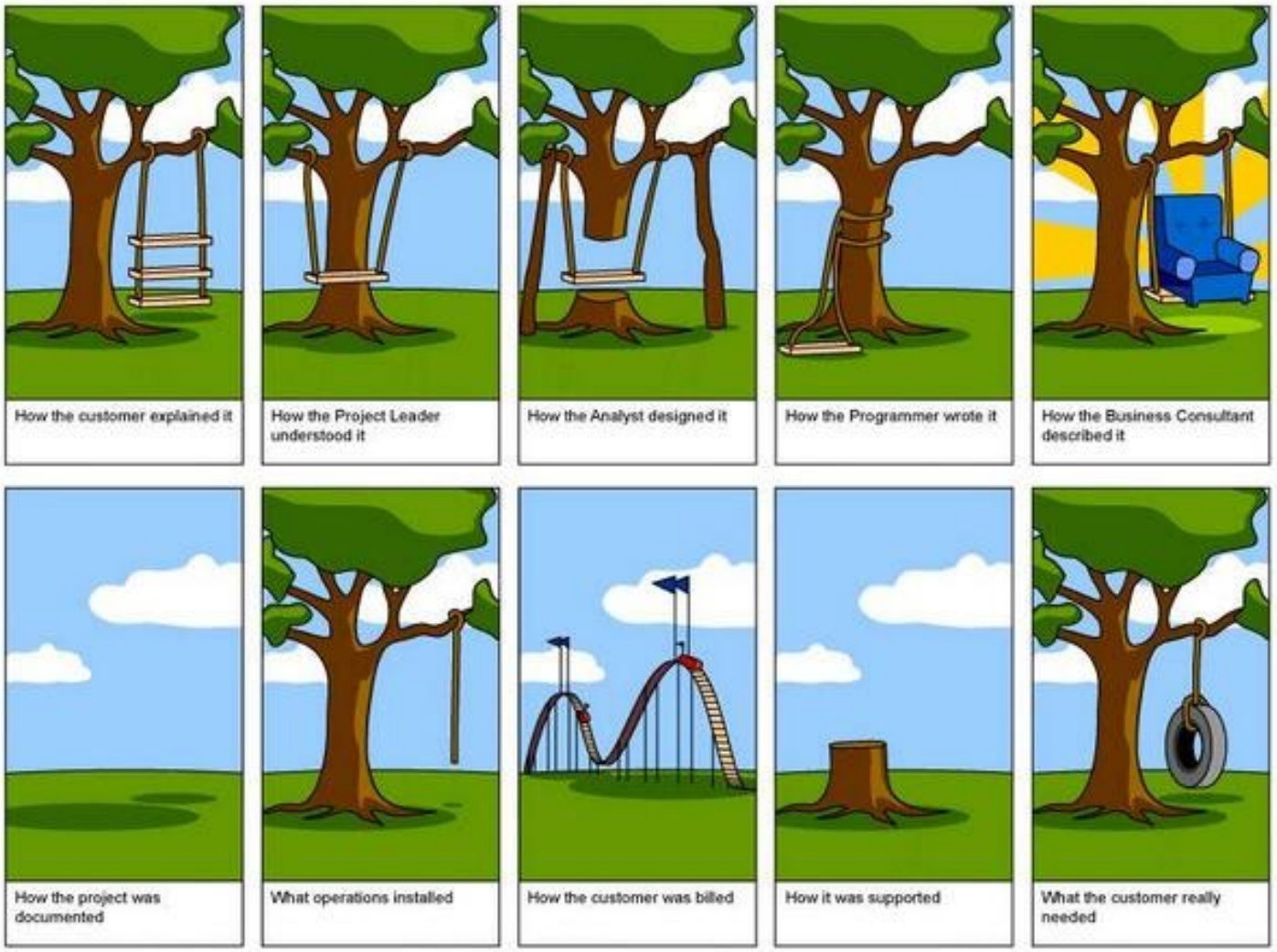
## Introducción a UML

- La búsqueda de una solución de una aplicación de software utilizando P.O.O. pasa por la construcción de un modelo. El modelo tiene la capacidad de abstraernos de los pequeños detalles y evitar la complicación del mundo real.
- Unified Modeling Language o UML es un lenguaje de modelado. UML nos ayuda en el ciclo de desarrollo de software (SDLC)



# Nociones de UML (II)

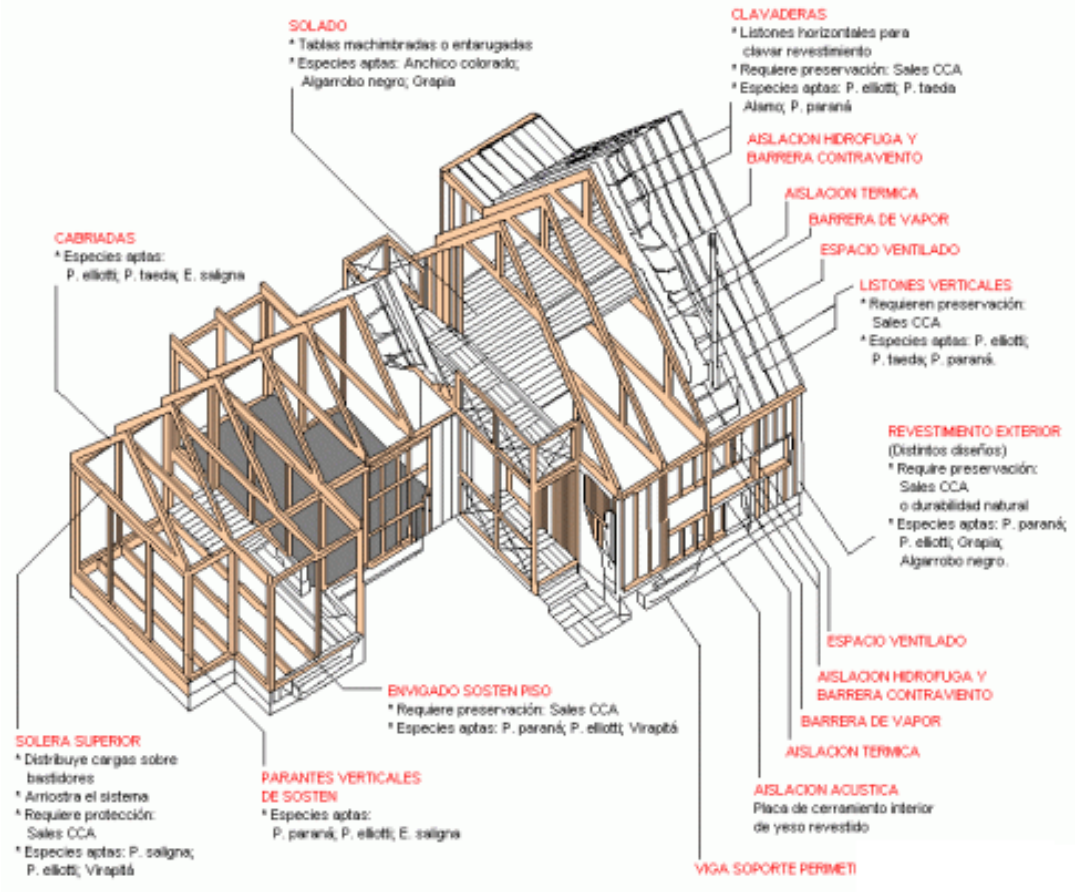
## Necesidad de un modelo



# Nociones de UML (III)

## Concepto de Modelo

- "Un modelo es una simplificación de la realidad" (Booch ). Es una conceptualización abstracta de algún tipo de entidad como un edificio o sistema.

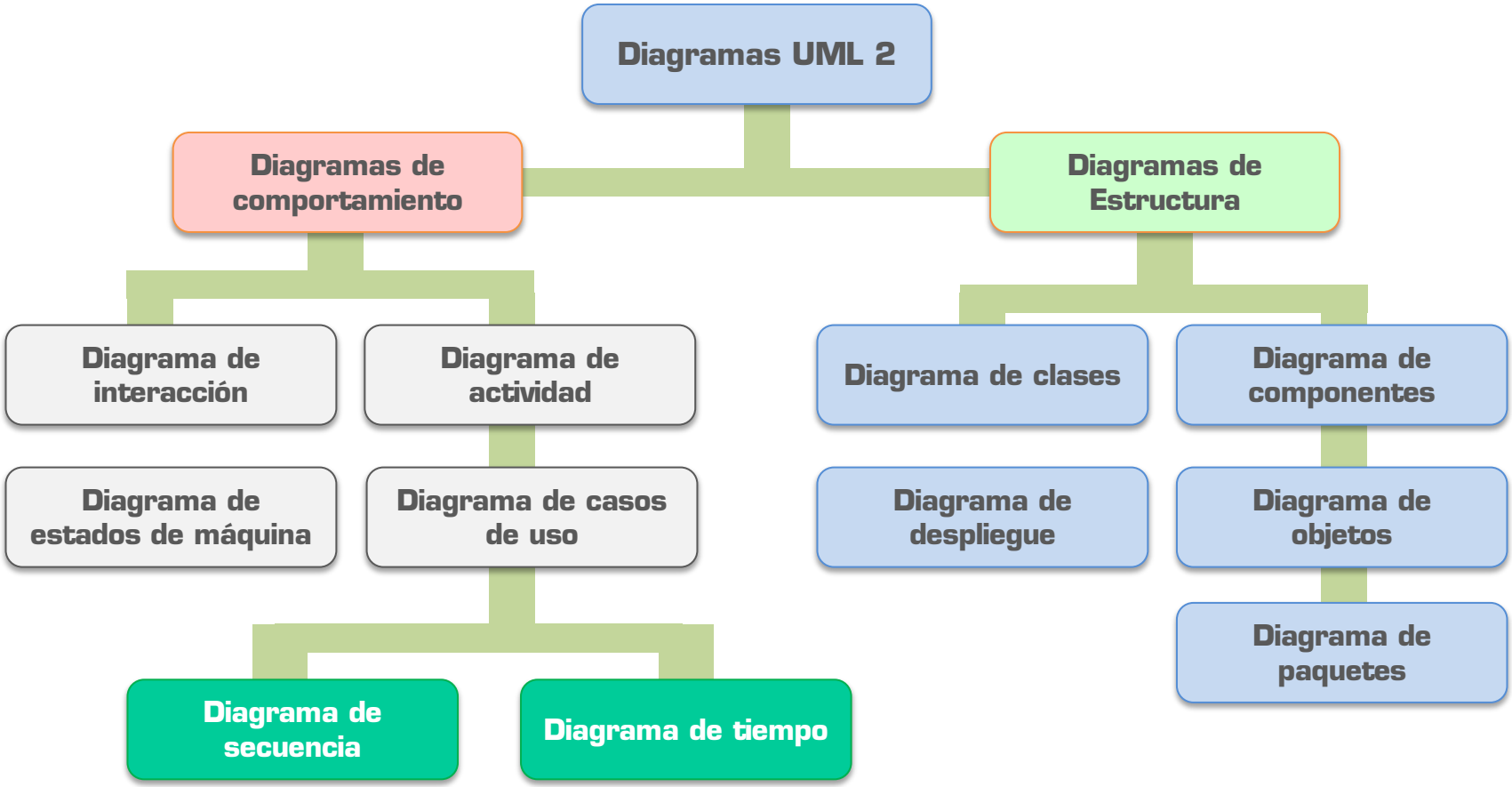


# Nociones de UML (IV)

## Concepto de Modelo

- Necesitamos crear modelos , porque nos permiten varias cosas, como:
  - **Visualizar o comprender sistemas nuevos o existentes.**
  - **Crear un sistema de comunicación que no se interprete de forma diferentes entre los integrantes del equipo.**
  - **Para documentar las decisiones realizadas en las fases de análisis del SDLC.**
  - **Decidir y concretar la estructura estática y el comportamiento dinámico de los elementos que integran el sistema.**
  - **Utilizar una plantilla para la construcción de la solución de software.**

# Nociones de UML (V): diagramas



# Nociones de UML (VI)

## Herramientas adicionales

- Además de las herramientas que nos proporciona UML, es recomendable usar los patrones. Existen varios catálogos de patrones:
- **Patrones GoF** (Gang of four), contiene 23 patrones estándar agnósticos a la arquitectura que se utilice.
- **Patrones de arquitectura**, contiene únicamente 2 patrones:
  - Layers
  - Model-View-Controller.
- Patrones J2EE, contiene los patrones específicos de la arquitectura Enterprise de Java.

# Notación gráfica: asociación

- Existen múltiples formas de asociación, cada una representada por una línea y por un icono en el extremo de ésta, indicando el tipo concreto de asociación que se crea entre ambas clases:

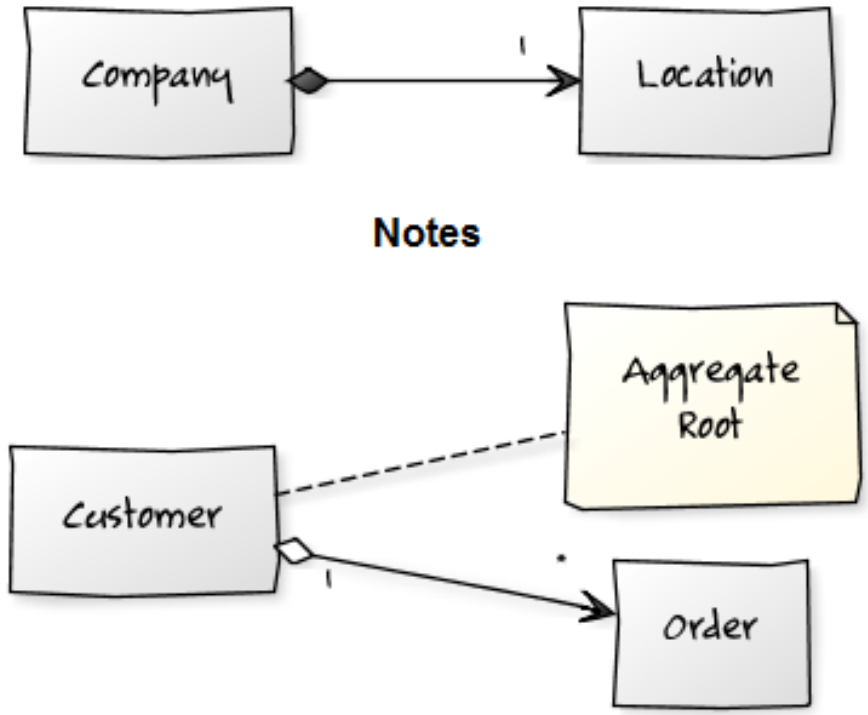

**Aggregation**


**Composition**

- Como agregación o composición.

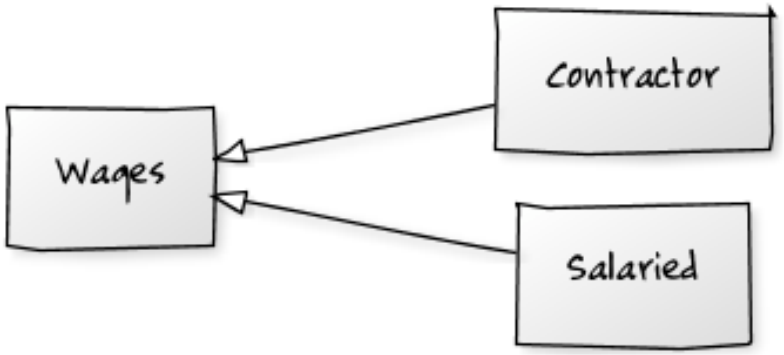
# Notación gráfica: herencia

- UML es un lenguaje de comunicación y sus gráficos intentan transmitir la captura de requerimientos, para que esta comunicación no se difumine, es necesario, ayudarse de elementos como notas que aclaren conceptos y gráficos:

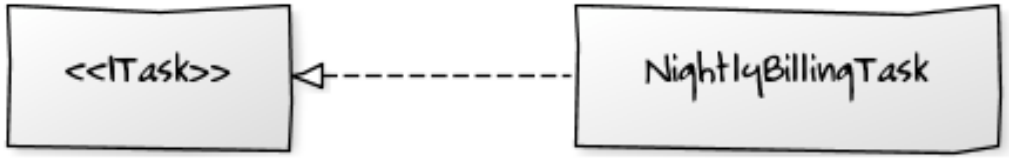


# Notación gráfica: herencia (y II)

- La herencia es un tipo de asociación especial, que aplica un acoplamiento muy alto entre clases y que deberíamos utilizar con mucho cuidado, evitando abusar del concepto:



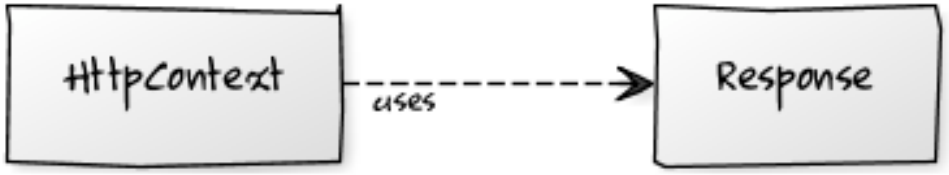

**Interface Inheritance**




**International Characters**

# Notación gráfica: interfaces

- Los interfaces son elementos muy importantes y juegan un papel vital en muchos frameworks actuales, además fomentan la aplicación de principios de arquitectura básicos, en UML se representan:

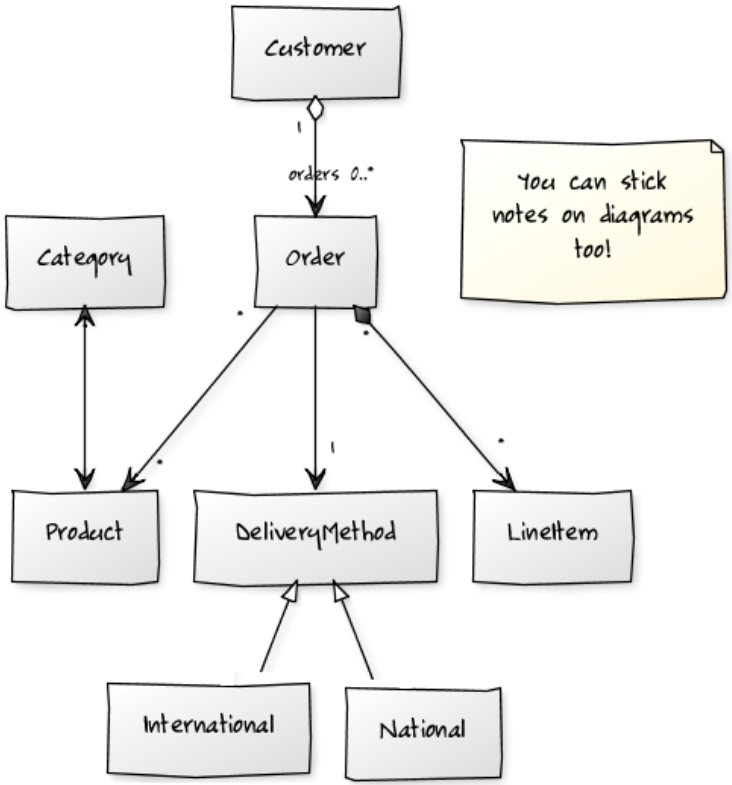


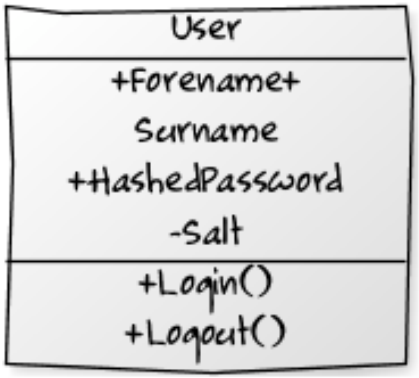

**Interface**


**Class with Details**

# Notación gráfica: interfaces (y II)

- Para lograr crear el modelo de dominio, es necesario realizar un exhaustivo análisis y crear el diagrama de clases con UML, que posteriormente podremos validar con el diagrama de objetos, la representación del primero es:



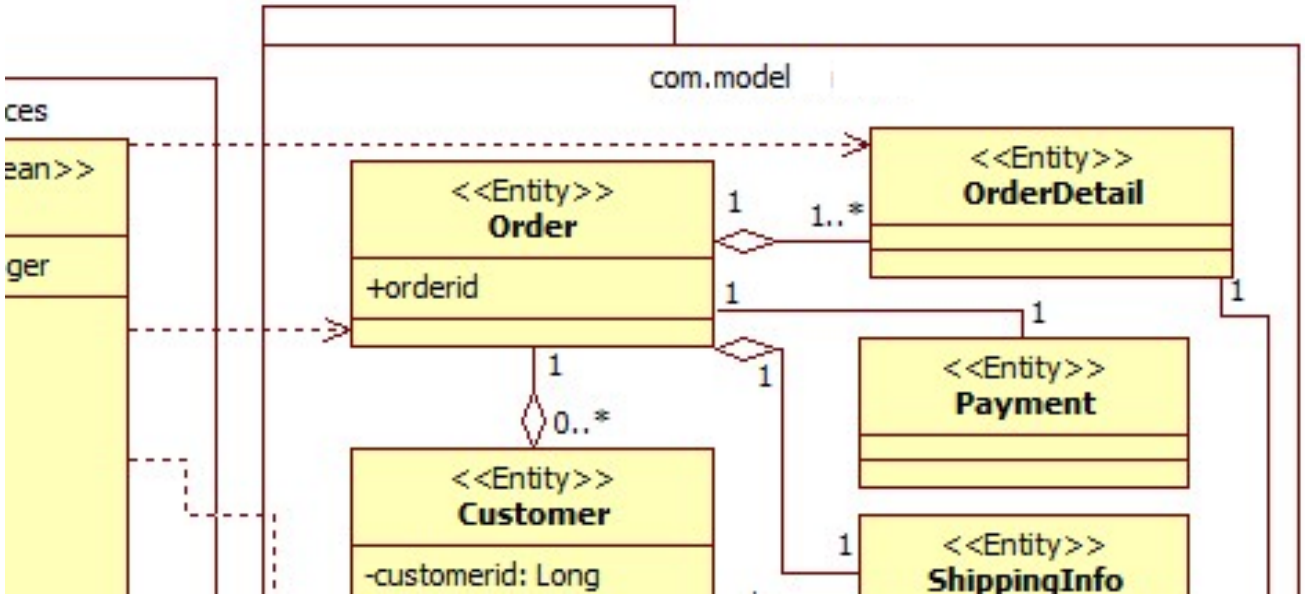



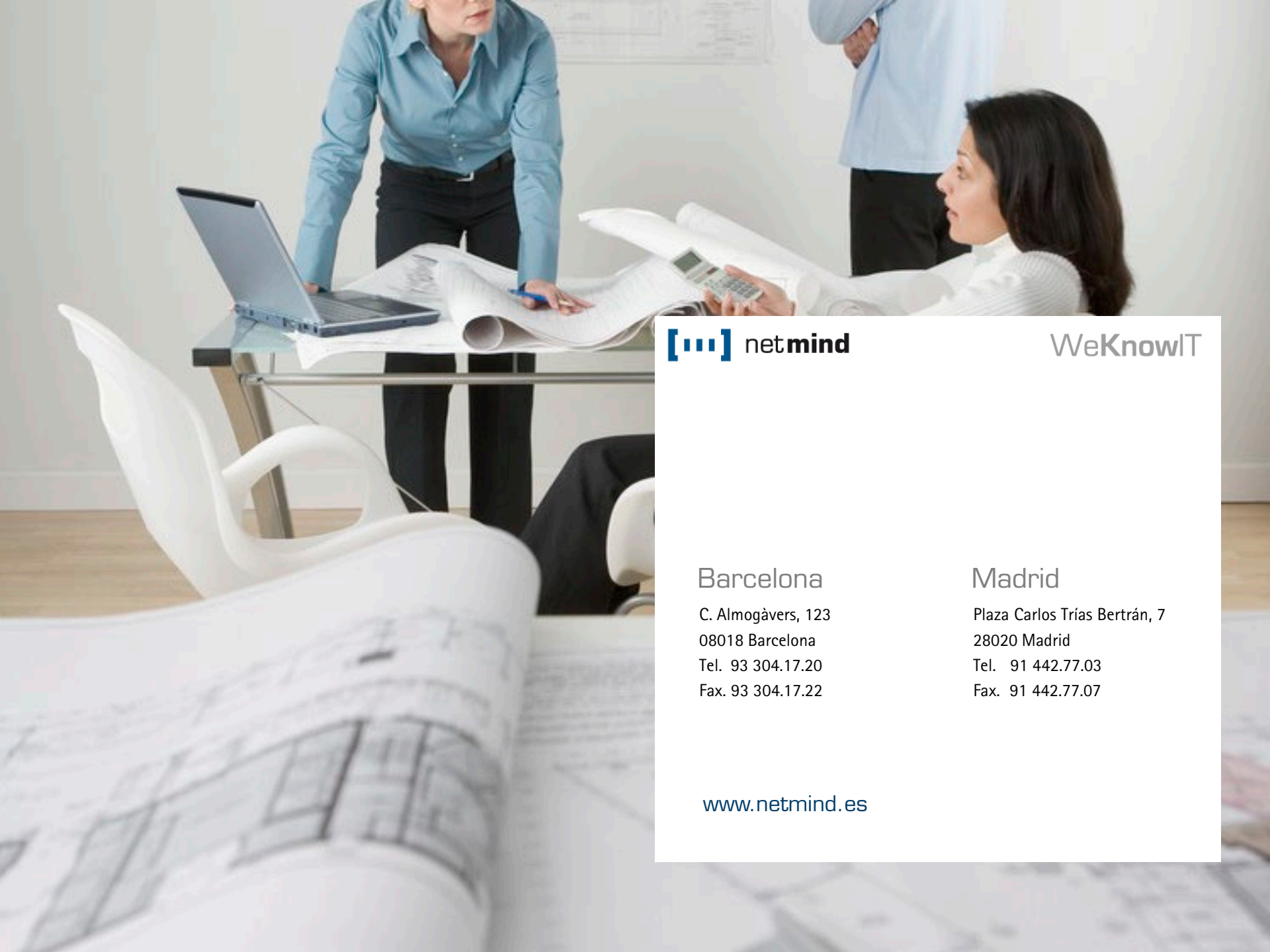

**Something Meaty!**

# Modelando la aplicación

## Modelo del dominio

- Hace referencia al **conjunto de clases necesarias para crear una aplicación** y que se **convierta** en **solución** de las necesidades del cliente, se representa por un **diagrama de clases** en formato UML.





netmind

WeKnowIT


## Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

## Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

[···] netmind

WeKnowIT

# MI1717 – Parte 10

## Introducción a la Programación Orientada a Objetos en Java

## Entornos de desarrollo

# Programación Orientada a Objetos
# Entornos de desarrollo

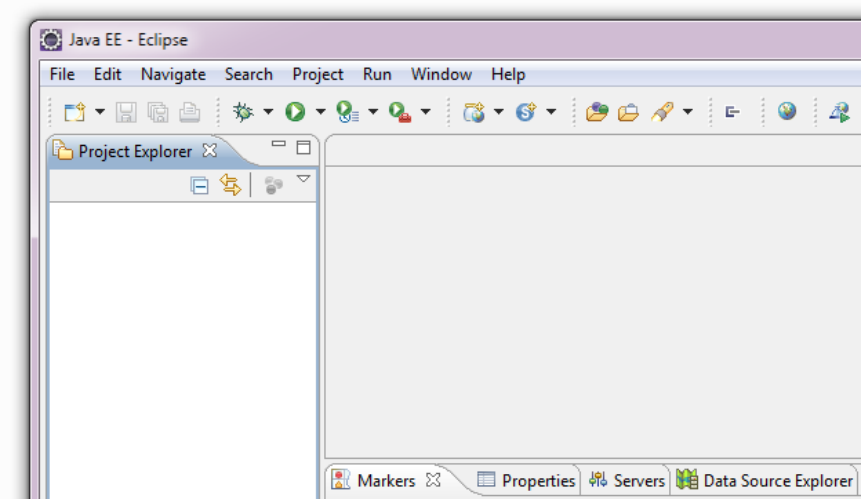
## Módulo 10

- Introducción a Eclipse
- Compilación y ejecución de una clase
- Depuración de una clase
- Uso del depurador
- Uso de bibliotecas externas
- Eclipse como plataforma
- Otros entornos Java: Netbeans, IntelliJ IDEA
- Entornos .NET: Visual Studio, Visual Studio Code

# Introducción

## Entorno de desarrollo integrado (IDE)

- Eclipse es un entorno de desarrollo integrado (IDE) que fue donado a la fundación eclipse (http://www.eclipse.org) por IBM basada en la herramienta de desarrollo WSAD.

# Introducción (II)

## Eclipse – Versiones

- 1.0 – Noviembre 2001
- Siempre se liberan versiones en junio (desde 2008 el 4º miércoles de junio)

| Nombre | Fecha | Versión |
|---|---|---|
| N/A | 21/06/2004 | 3.0 |
| N/A | 28/06/2005 | 3.1 |
| Callisto | 30/06/2006 | 3.2 |
| Europa | 29/06/2007 | 3.3 |
| Ganymede | 25/06/2008 | 3.4 |
| Galileo | 24/06/2009 | 3.5 |
| Helios | 23/06/2010 | 3.6 |

| Nombre | Fecha | Versión |
|---|---|---|
| Indigo | 22/06/2011 | 3.7 |
| Juno | 27/06/2012 | 3.8 y 4.2 |
| Kepler | 26/06/2013 | 4.3 |
| Luna | 25/06/2014 | 4.4 |
| Mars | 24/06/2015 | 4.5 |
| Neon | 22/06/2016 | 4.6 |
| Oxygen | 21/06/2017 | 4.7 |

# Introducción (y III)

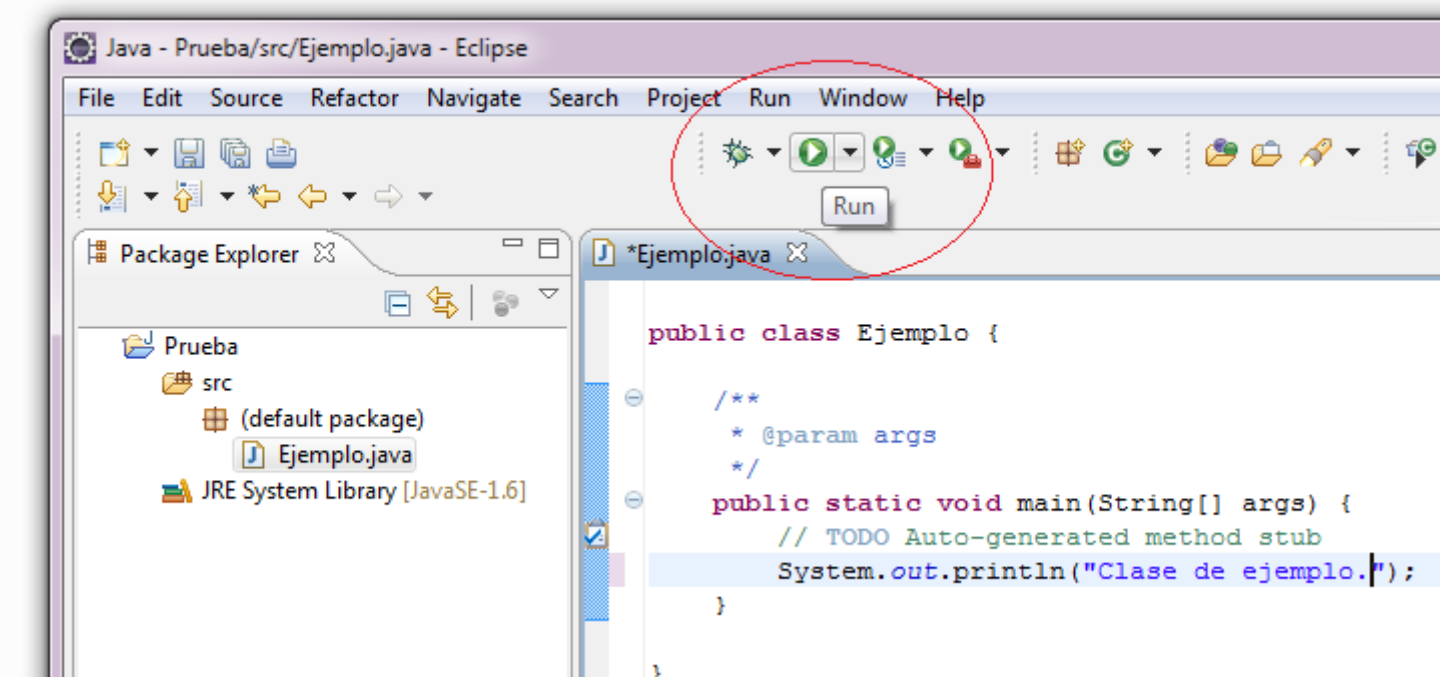
## Eclipse

- Eclipse se distribuye con un gran conjunto de plugins (algunos de ellos sólo disponibles en WSAD).
- Requiere un **JDK** acorde a la versión en la que queramos desarrollar, podemos incorporar un servidor Java EE para desarrollar aplicaciones web. Proporciona soporte para las páginas JavaServer Faces. Eclipse utiliza el concepto de vistas y perspectivas.
- Una vista es una paleta concreta para ser usada con una tecnología determinada como base de datos.
- Una perspectiva es un conjunto de paletas seleccionadas por WebSphere para poder trabajar con Java SE, Java EE, por ejemplo.

# Compilación y ejecución de una clase

## Ejecutar un programa / aplicación

- Para compilar un programa, sólo es necesario guardar los cambios de la clase en la que estemos trabajando (por ejemplo, haciendo clic en el icono del disquete) y para ejecutarla debemos hacer clic en el icono del play (señalado en rojo en la figura inferior).



# Compilación y ejecución de una clase (II)

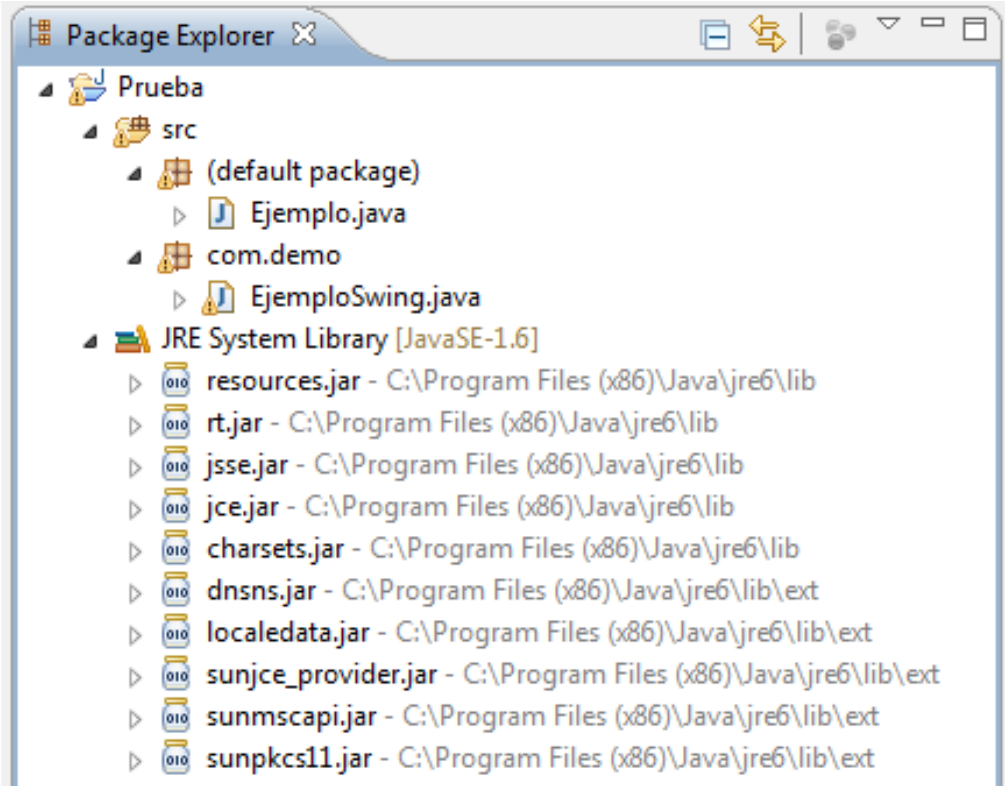
## Editor de código

- La pantalla de código fuente, incorpora múltiples mecanismos y utilidades que facilitan la creación del código, como por ejemplo los atajos

# Compilación y ejecución de una clase (III)

## Vista de proyectos: Package Explorer

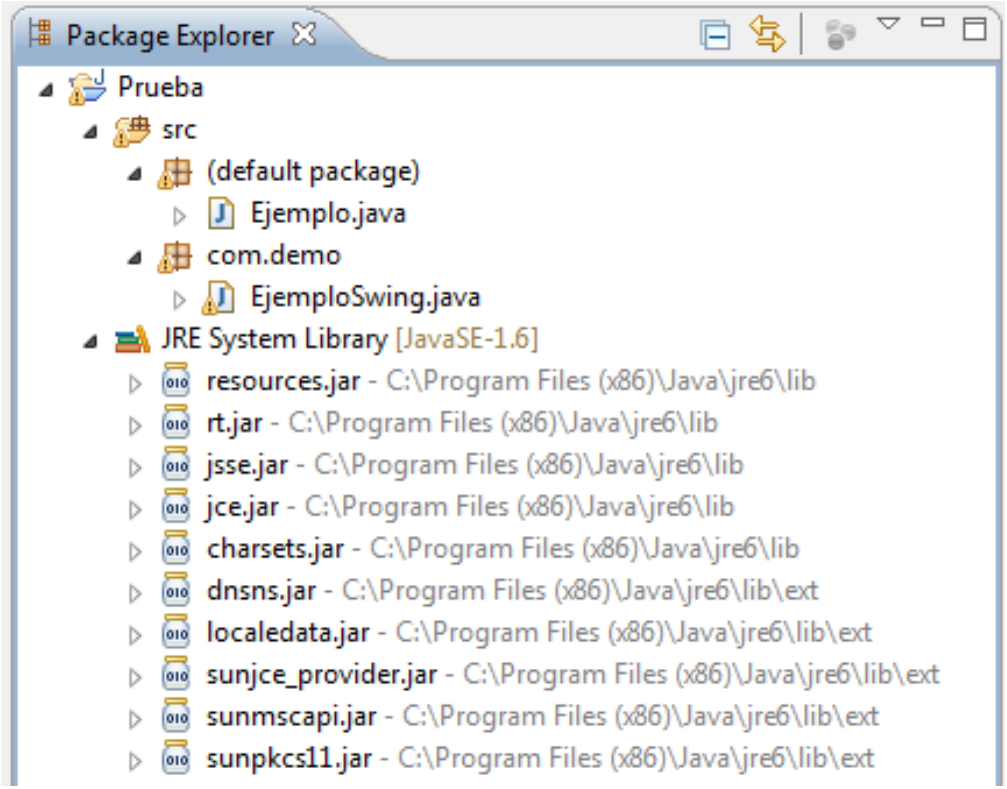
- Cada vez que creamos un proyecto en Eclipse se añade a la vista de proyectos, desde ella podemos ver una representación jerárquica del contenido del proyecto organizado en carpetas



# Compilación y ejecución de una clase (III)

## Vista de proyectos: Package Explorer

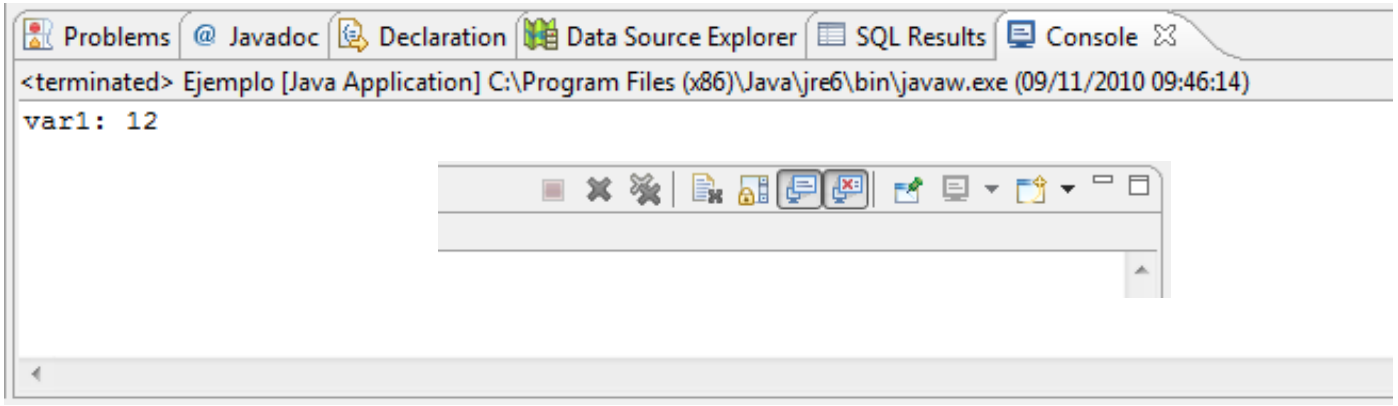
- Cada vez que creamos un proyecto en Eclipse se añade a la vista de proyectos, desde ella podemos ver una representación jerárquica del contenido del proyecto organizado en carpetas



# Compilación y ejecución de una clase (IV)

## Vista de salida de datos: Consola

- Al ejecutar una clase o una aplicación todas las invocaciones a System.out.print salen por la Consola, como si se tratase de una ventana del sistema operativo.

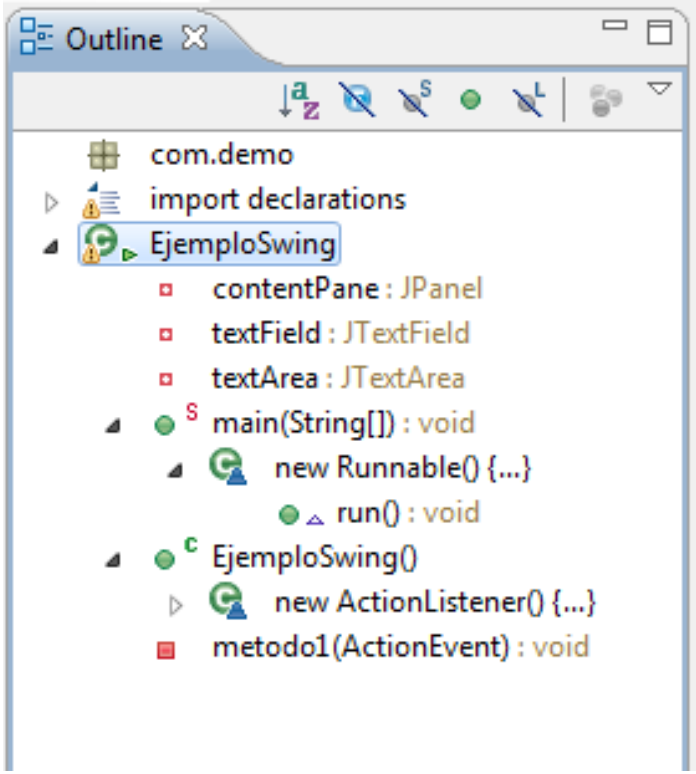



- En la parte superior derecha de la Consola, podemos encontrar iconos que nos permiten limpiar la consola, acceder a otra consola de otra aplicación, y controlar su visibilidad y apariencia.

# Compilación y ejecución de una clase (V)

## Vista del contenido de una clase: Outline

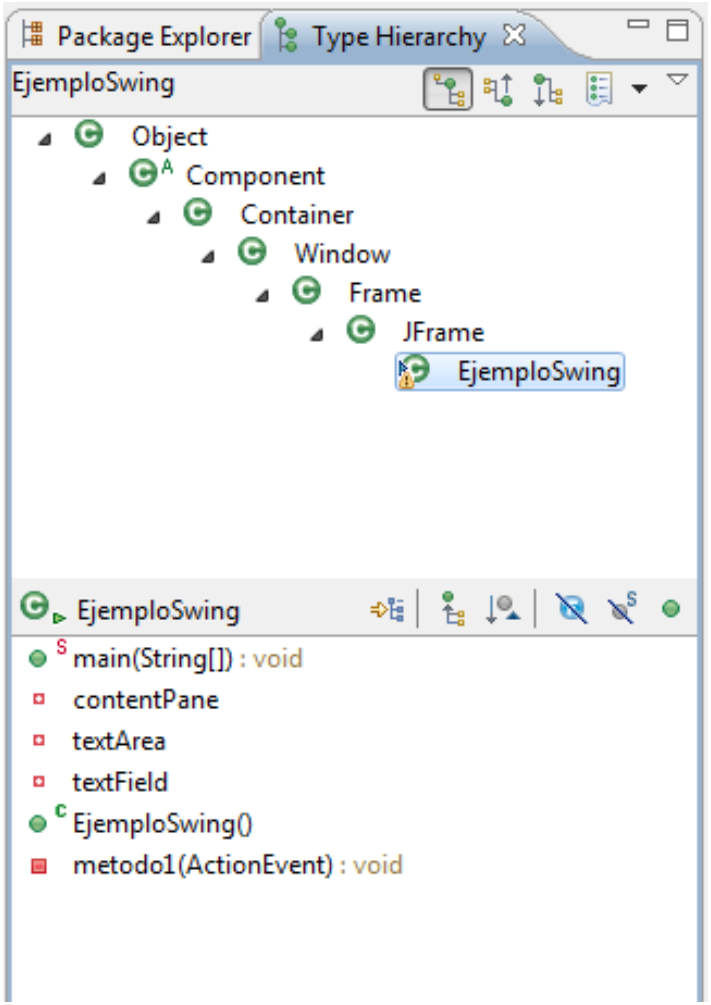
- La vista Outline es muy práctica, sirve para ver los elementos de una clase, es decir, sus propiedades y sus métodos con los iconos que nos indican su visibilidad, nivel de protección, si son estáticos, etc..
- Si hacemos doble clic sobre un elemento como un método, Eclipse sitúa automáticamente el cursor en la línea correspondiente al elemento marcado.
- Con el botón derecho del ratón, podemos acceder a un buen número de opciones.



# Compilación y ejecución de una clase (y VI)

## Vista del contenido de una clase: Type Hierarchy

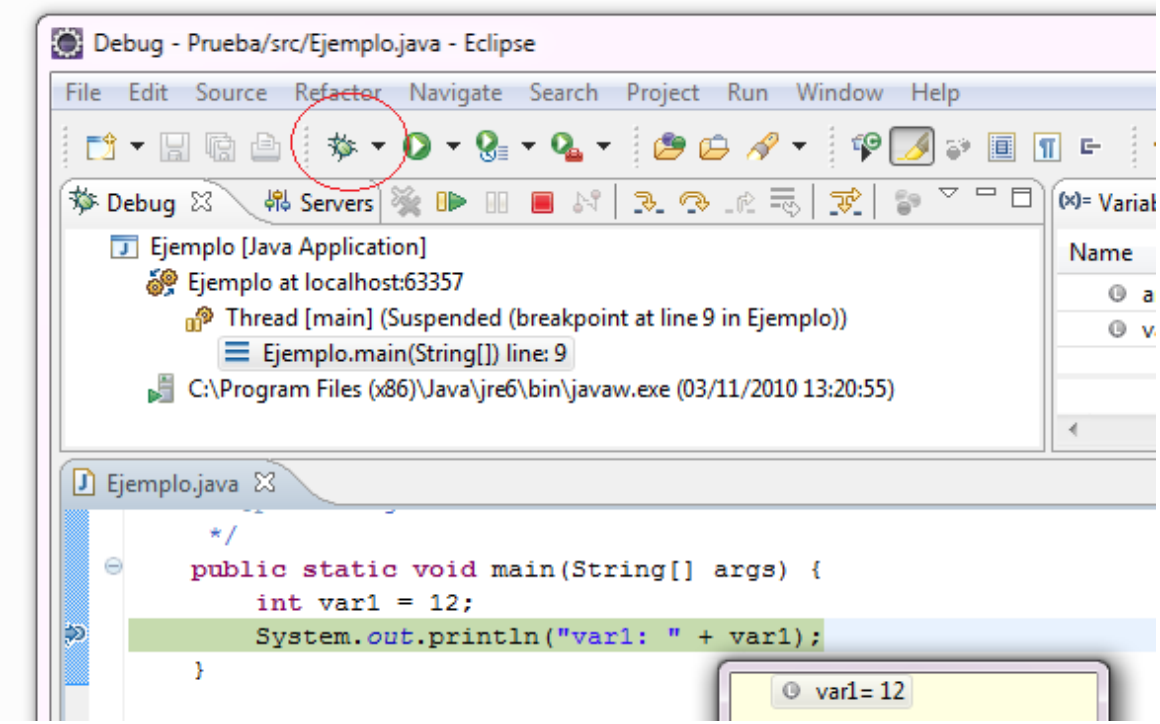
- Con la vista Type Hierarchy, podemos mostrar la jerarquía de cualquier clase Java.
- Debemos arrastrar una clase seleccionada en la vista Outline y soltarla sobre la vista Type Hierarchy.
- Esta vista está formada por dos paleta, en la parte superior, se muestra la ruta de herencia de la clase, mientras que en la paleta inferior, se muestra el contenido de la clase que hemos seleccionado haciendo clic sobre ella.



# Depuración de una clase

## Ejecutar un programa / aplicación

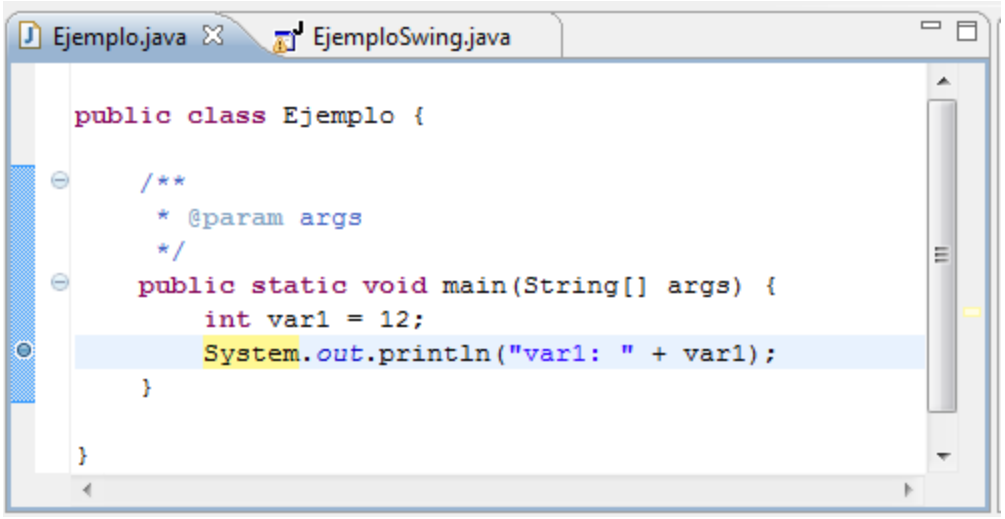
- Para depurar una clase, se establece un breakpoint en la barra vertical y pulsamos sobre debug (señalada en le imagen), podemos ver los valores de las variables y avanzar paso a paso, tal como se explica en las siguientes diapositivas.



# Uso del depurador

## Depuración de código – 1er paso

- Durante el desarrollo de una aplicación, es frecuente encontrarse con la necesidad de llevar a cabo tareas de depuración. Para depurar un trozo de código, es necesario, en primer lugar, establecer breakpoints.

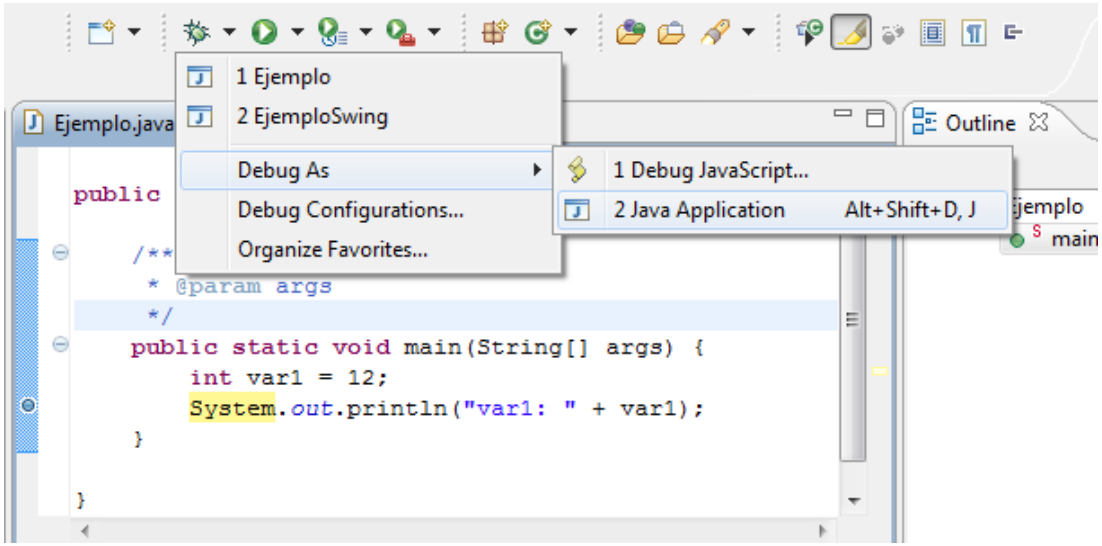



- Para establecer un breakpoint, haga doble clic en la barra vertical de color gris situada a la izquierda del panel de código hasta que aparezca un círculo de color verde señalando la línea en deberá pararse la ejecución.

# Uso del depurador (II)

## Depuración de código – 2do paso

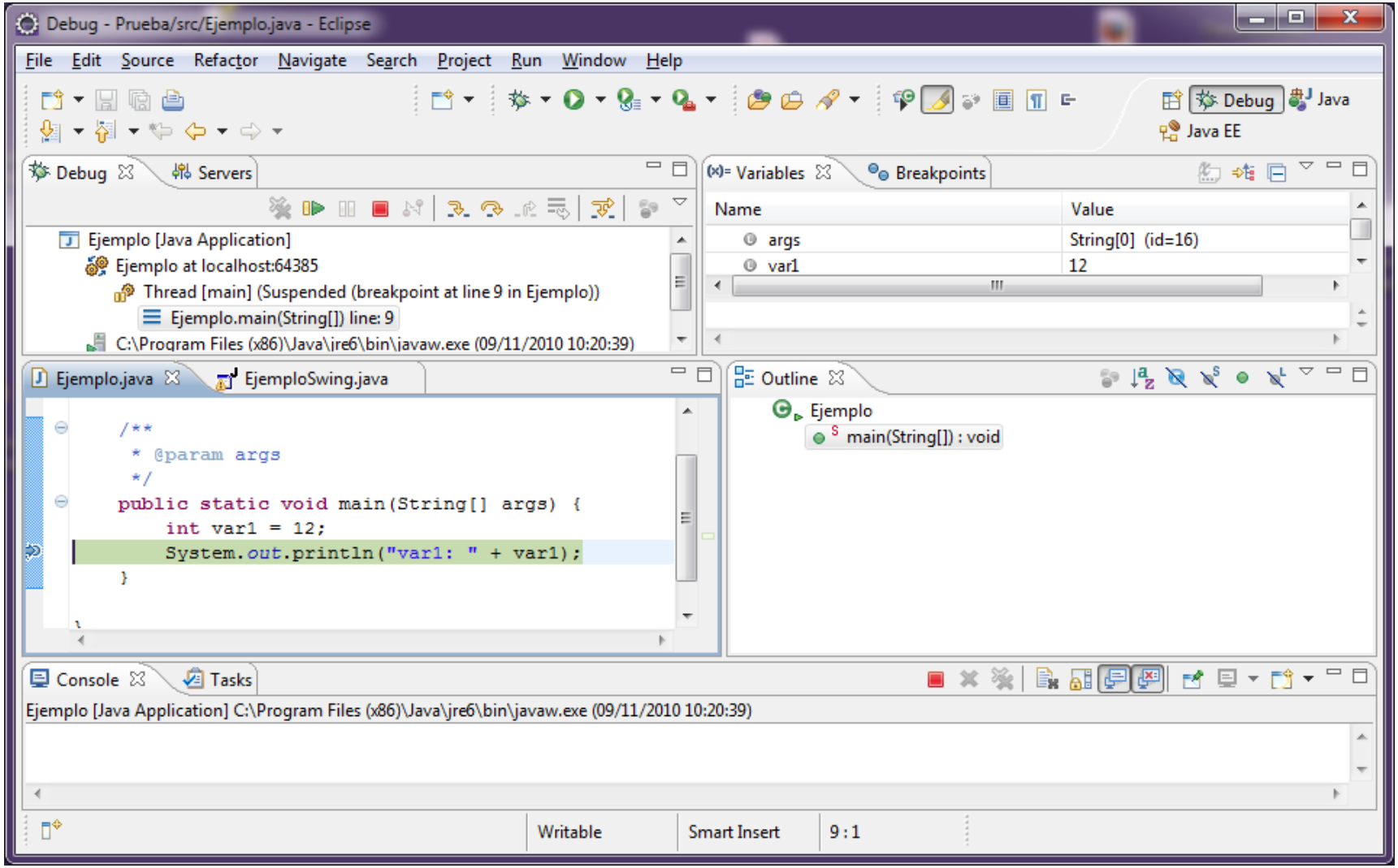
- Una vez establecido el breakpoint, debemos ejecutar en modo de depuración el código, para ello, hacemos clic en el icono con una cucaracha y seleccionamos Debug as >> Java Application, tal como muestra la siguiente figura:




- Eclipse nos indica que es necesario un cambio de perspectiva que nosotros confirmamos.

# Uso del depurador (III)

## Depuración de código – 3er paso



# Uso del depurador (IV)

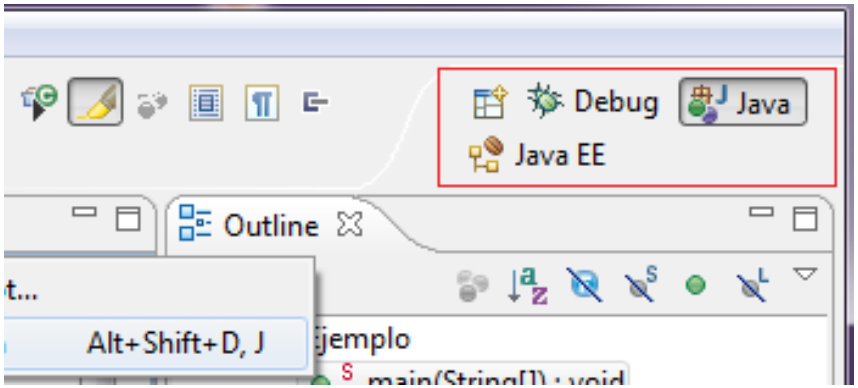
**Vistas y perspectivas**

- Eclipse utiliza el concepto de trabajo de vistas y perspectivas. Una vista es una paleta concreta, por ejemplo, la Consola.
- Un conjunto de vistas (paletas) es una perspectiva, podemos añadir vistas a una perspectiva con el comando **Window >> Show View** y seleccionando la que deseemos.
- También podemos reclamar una perspectiva de las que Eclipse / WSAD incorporan con el comando **Window >> Open perspective** y seleccionando la que deseemos.
- Finalmente podemos personalizar una perspectiva con el comando **Window >> Customize perspective** y seleccionando en el cuadro de diálogo las vistas que formarán parte de la perspectiva actual.

# Uso del depurador (y V)

**Perspectivas**

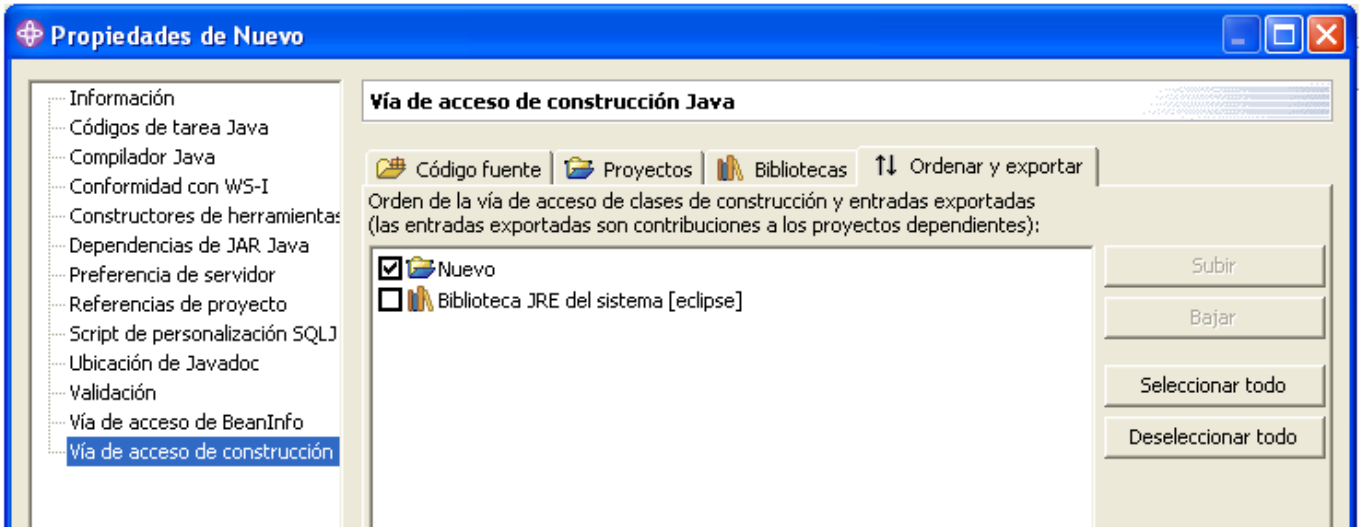
- Al principio de usar Eclipse es posible que nos desaparezcan vistas o que estemos en una perspectiva y no encontremos una determinada vista. Podemos restablecer el entorno con el comando **Window >> Reset perspective**.
- Cada perspectiva abierta dispone de un icono asociado que a medida que abrimos perspectiva se van "apilando" en la parte superior derecha del IDE, podemos pasar de una perspectiva a otra haciendo clic en el icono correspondiente.



# Uso de bibliotecas externas

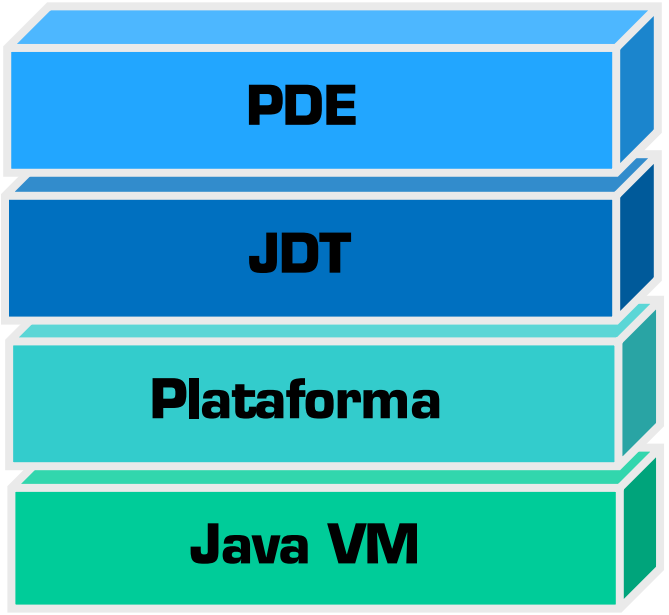
## Propiedades del proyecto

- La pantalla de propiedades, reúne toda la configuración del proyecto actual.
- Esta pantalla es el lugar idóneo, desde el cual deberemos añadir las dependencias de nuestro proyecto al incorporar bibliotecas. Podemos definir propiedades para el entorno de ejecución, web services, javadoc…



# Eclipse como plataforma

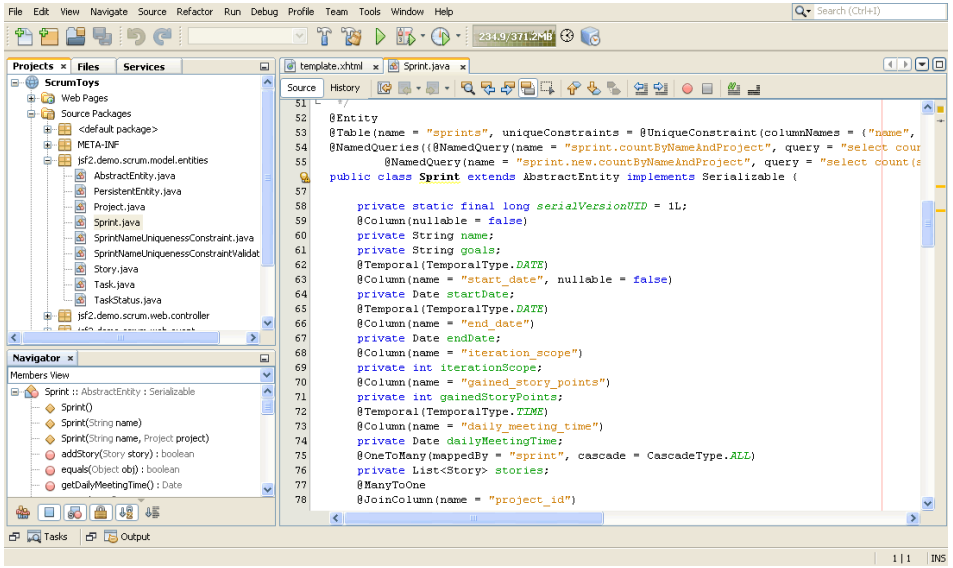
- Eclipse es una plataforma universal para integrar herramientas de desarrollo, cuanta con un framework RCP para generar aplicaciones basadas en el.
- La arquitectura de eclipse está basada fundamentalmente en 4 capas:
  - Plugins para el IDE
  - Herramientas de desarrollo Java
  - Plataforma de eclipse
  - Máquina virtual de Java



# Otros entornos – NetBeans

## NetBeans

- NetBeans fue creado originalmente como Xelfi por Roman Stanek, posteriormente vendido a Sun Microsystems (1999) y desde 2010 es propiedad de Oracle Corporation

# Otros entornos – NetBeans (y II)

## NetBeans – Versiones

- 1.0 – 1997
- DeveloperX2 – 1999

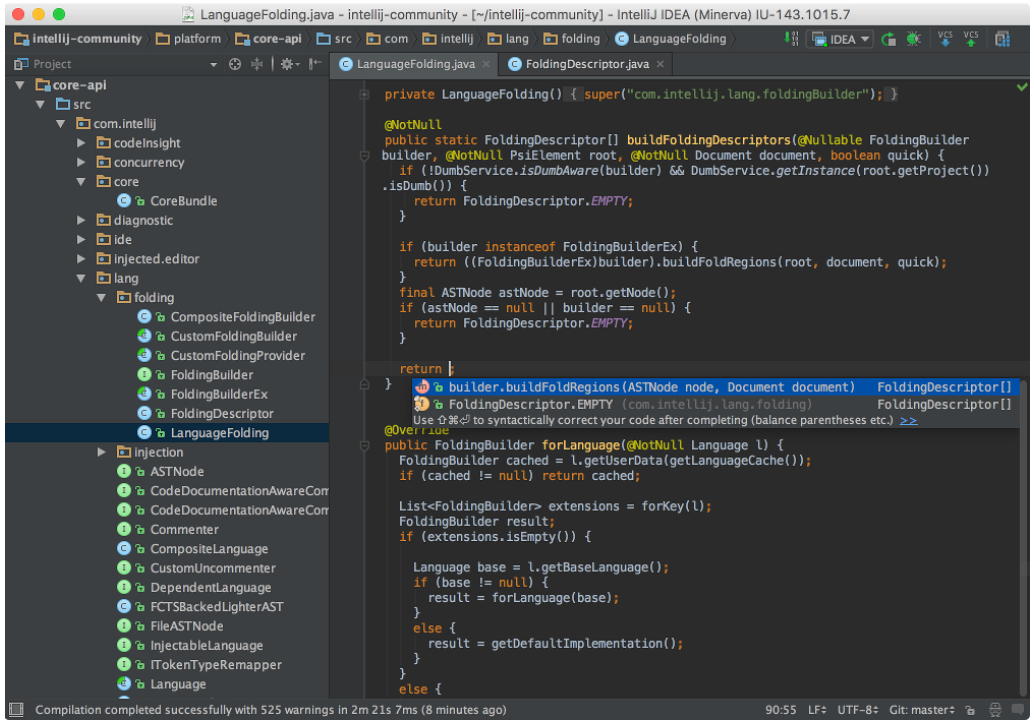
| Fecha | Versión |
|---|---|
| Enero 2003 | 3.5 |
| Abril 2004 | 3.6 |
| Diciembre 2004 | 4.0 |
| Mayo 2005 | 4.1 |
| Enero 2006 | 5.0 |
| Octubre 2006 | 5.5 |
| Diciembre 2007 | 6.0 |

| Fecha | Versión |
|---|---|
| Noviembre 2008 | 6.5 |
| Junio 2010 | 6.9 |
| Abril 2011 | 7.0 |
| Marzo 2014 | 8.0 |
| Noviembre 2015 | 8.1 |
| Octubre 2016 | 8.2 |
| Julio-Agosto 2017 | 9.0 |

# Otros entornos – IntelliJ IDEA

## IntelliJ IDEA

- IntelliJ IDEA fue creado por JetBrains en 2001
- Tiene una versión community y una versión comercial propietaria
- Android Studio está construida sobre IntelliJ IDEA



# Otros entornos – IntelliJ IDEA (y II)

**IntelliJ IDEA – Versiones**

- 1.0 – 1997
- Las versiones van numeradas en formato año.release.revisión
- La última versión es la 2016.3.2 – Noviembre 2016

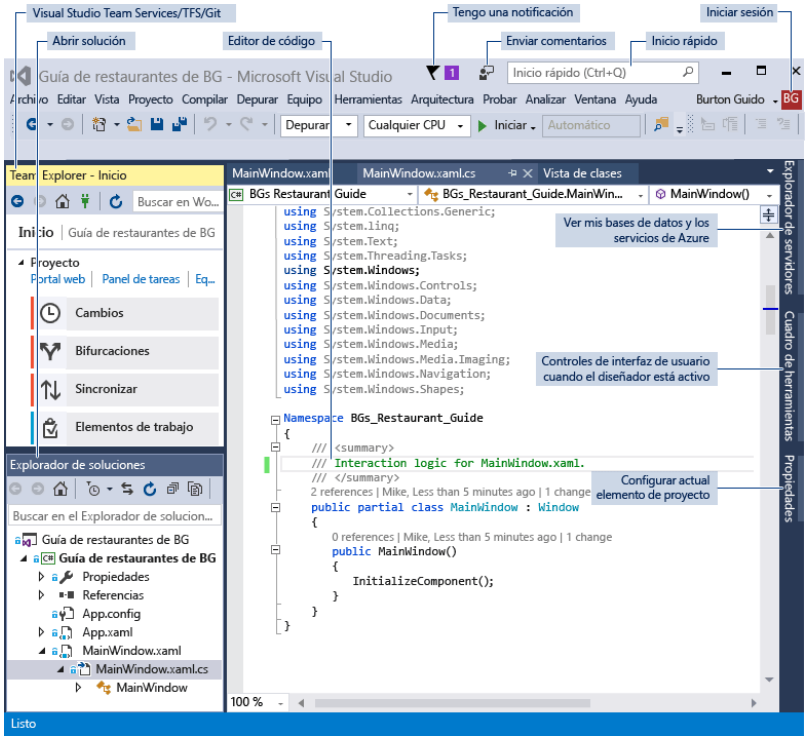
- Descarga: https://www.jetbrains.com/idea/

- Versiones: https://www.jetbrains.com/idea/#chooseYourEdition

# Otros entornos – Visual Studio

## Microsoft Visual Studio

- Visual Studio nació en 1997 al unir Visual Basic 3, Visual C++, Visual FoxPro
- Cuenta con una versión Community gratuita
- Ofrece dos ediciones Professional y Enterprise comerciales



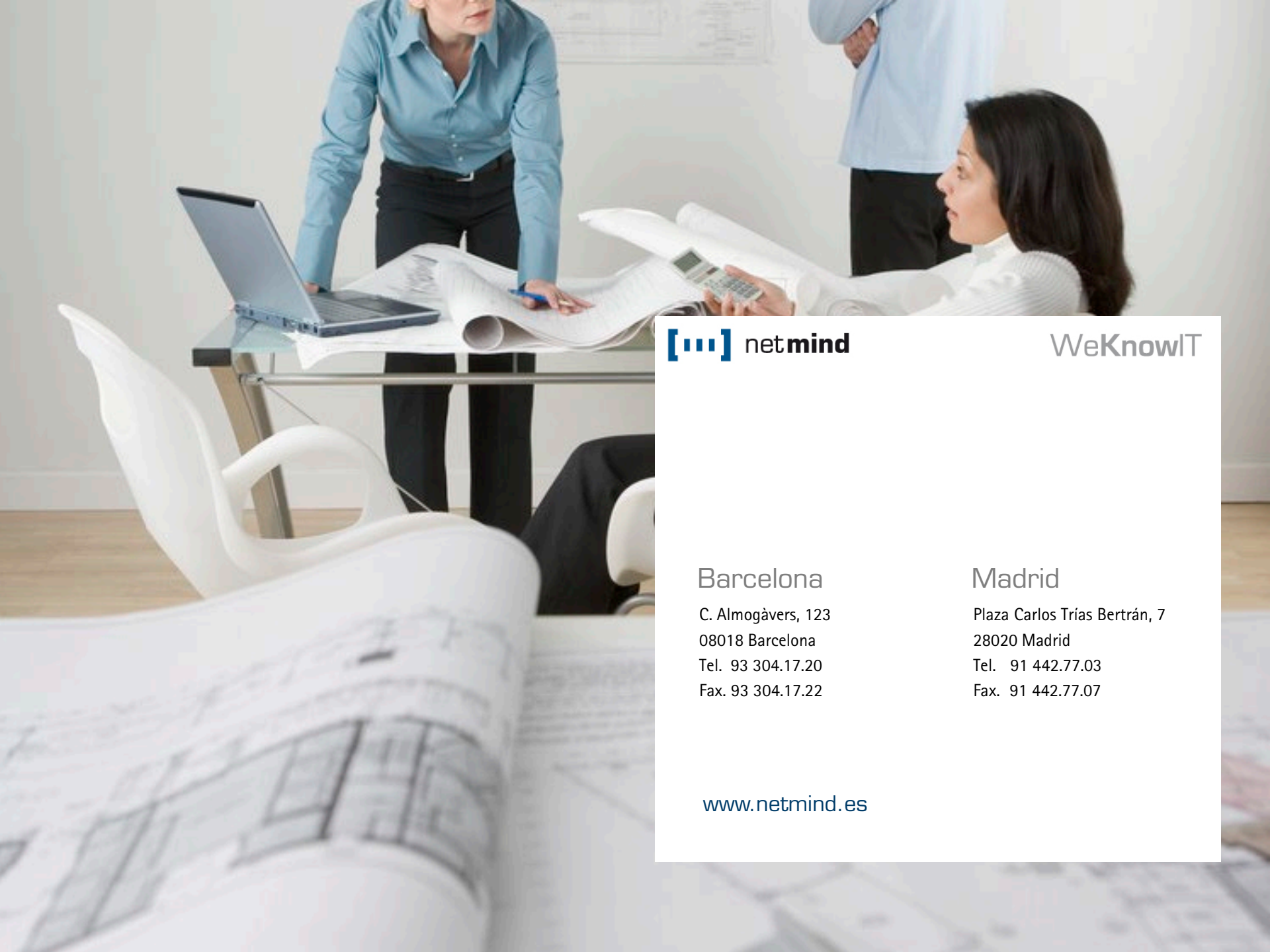
# Otros entornos – Visual Studio (y II)

## Microsoft Visual Studio – Versiones

- Descarga: https://www.visualstudio.com/es/downloads/

| Nombre | Fecha | Versión |
|---|---|---|
| Visual Studio | Abril de 1995 | 4.0 |
| Visual Studio 97 | Febrero de 1997 | 5.0 |
| Visual Studio 6.0 | Junio de 1998 | 6.0 |
| Visual Studio .NET (2002) | 13/02/2002 | 7.0 |
| Visual Studio .NET 2003 | 24/04/2003 | 7.1 |
| Visual Studio 2005 | 07/11/2005 | 8.0 |

| Nombre | Fecha | Versión |
|---|---|---|
| Visual Studio 2008 | 19/11/2007 | 9.0 |
| Visual Studio 2010 | 12/04/2010 | 10.0 |
| Visual Studio 2012 | 12/11/2012 | 11.0 |
| Visual Studio 2013 | 17/10/2013 | 12.0 |
| Visual Studio 2015 | 20/07/2015 | 14.0 |
| Visual Studio 2017 | 2017 | 15.0 |



netmind

WeKnowIT


## Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

## Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es

Tú opinión es importante para nosotros

Por favor, envíanos tus comentarios y sugerencias sobre el contenido …

Desde Netmind agradecemos tu tiempo y tendremos en consideración tus comentarios para mejorar continuamente nuestra oferta formativa para que siempre os encontréis satisfechos de habernos elegido.

quality@netmind.es

**Barcelona** Carrer Almogàvers 123 (Edificio Ecourban) / T. +34 933 041 720 / F. +34 933 041 722 / 08018 Barcelona
**Madrid** Plaza de Carlos Trías Bertrán 7, 1ª Planta (Edificio Sollube) / T. +34 914 427 703 / 28020 Madrid
**Atlanta** 296 South Main Street, Suite 300/ T. +1 (404) 678 366 1363 / Alpharetta, Atlanta, GA 30009-1973

info@netmind.es | www.netmind.es | www.netmind.net

## Training Areas

Project Management
Business Analysis Agile
Management
IT Service Management
Enterprise Architecture
Business Games
Digital Innovation
Transformative Leadership

www.netmind.es