

Programación Orientada a Objetos

POO con JAVA

Conceptos de la POO

- La programación orientada a objetos establece un equilibrio entre la importancia de los procesos y los datos
- La *abstracción* es un proceso mental de extracción de las características esenciales, ignorando los detalles superfluos
- La *encapsulación* es ocultar los detalles que dan soporte a un conjunto de características esenciales de una abstracción. Existirán dos partes, una visible que todos tienen acceso y se aporta la funcionalidad, y una oculta que implementa los detalles internos
- La *modularidad* es descomponer un sistema en un conjunto de partes
 - El **acoplamiento** entre dos módulos mide el nivel de asociación entre ellos; nos interesa buscar módulos poco acoplados
 - La **cohesión** de un módulo mide el grado de conectividad entre los elementos que los forman; nos interesa buscar una cohesión alta
- La *jerarquía* es un proceso de estructuración de varios elementos por niveles

Elementos de la POO

■ Clases y Objetos

- Una clase describe las estructuras de datos que lo forman y las funciones asociadas con él. Una clase es un modelo con el que se construyen los objetos
- Un objeto es un ejemplar concreto de una clase, que se estructura y comporta según se definió en la clase, pero su estado es particular e independiente del resto de ejemplares. Al proceso de crear un objeto se le llama generalmente instanciar una clase
- Las clases asumen el principio de encapsulación, se describe una vista pública que representa la funcionalidad de la misma, y una vista privada que describe los detalles de implementación
- Una clase es el único bloque de construcción, y por lo tanto, en una aplicación Java sólo hay clases; no existen datos sueltos ni procedimientos

Elementos de la POO

■ Atributos y estado

- Un atributo es cada uno de los datos de una clase que la describen; no incluyen los datos auxiliares utilizados para una implementación concreta
- Un atributo es una variable de instancia
- El estado de un objeto es el conjunto de valores de sus atributos en un instante dado

Elementos de la POO

■ Métodos y mensajes

- Un método define una operación sobre un objeto. En general, realizan dos posibles acciones: consultar el estado del objeto o modificarlo. Los métodos disponen de parámetros que permiten delimitar la acción del mismo
- Nos podemos encontrar con diversos tipos de métodos:
 - Consultan o modifican un atributo, normalmente nos referenciaremos a ellos como: getters & setters
 - Realizan operaciones sobre el conjunto de atributos, calculando valores o realizando modificaciones
 - Inician los atributos al principio del ciclo de vida, o liberan los recursos al final del ciclo; nos referiremos a ellos como constructores o destructores
- Un mensaje es la invocación de un método de un objeto. Podemos decir que un objeto lanza un mensaje (quien realiza la invocación) y otro lo recibe (el que ejecuta el método)
- Podemos rescribir que una clase es la descripción e implementación de un conjunto de atributos y métodos

Elementos de la POO

■ Herencia y polimorfismo

- La herencia es una característica que permite a las clases definirse a partir de otras, y así reutilizar su funcionalidad
- A la clase padre se le llama superclase, clase base...
- A la hija subclase, clase derivada....
- El polimorfismo es la capacidad de que un mismo mensaje funcione con diferentes objetos. Es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo concreto de objetos sobre el que se trabaja. El método opera sobre un conjunto de posibles objetos compatibles

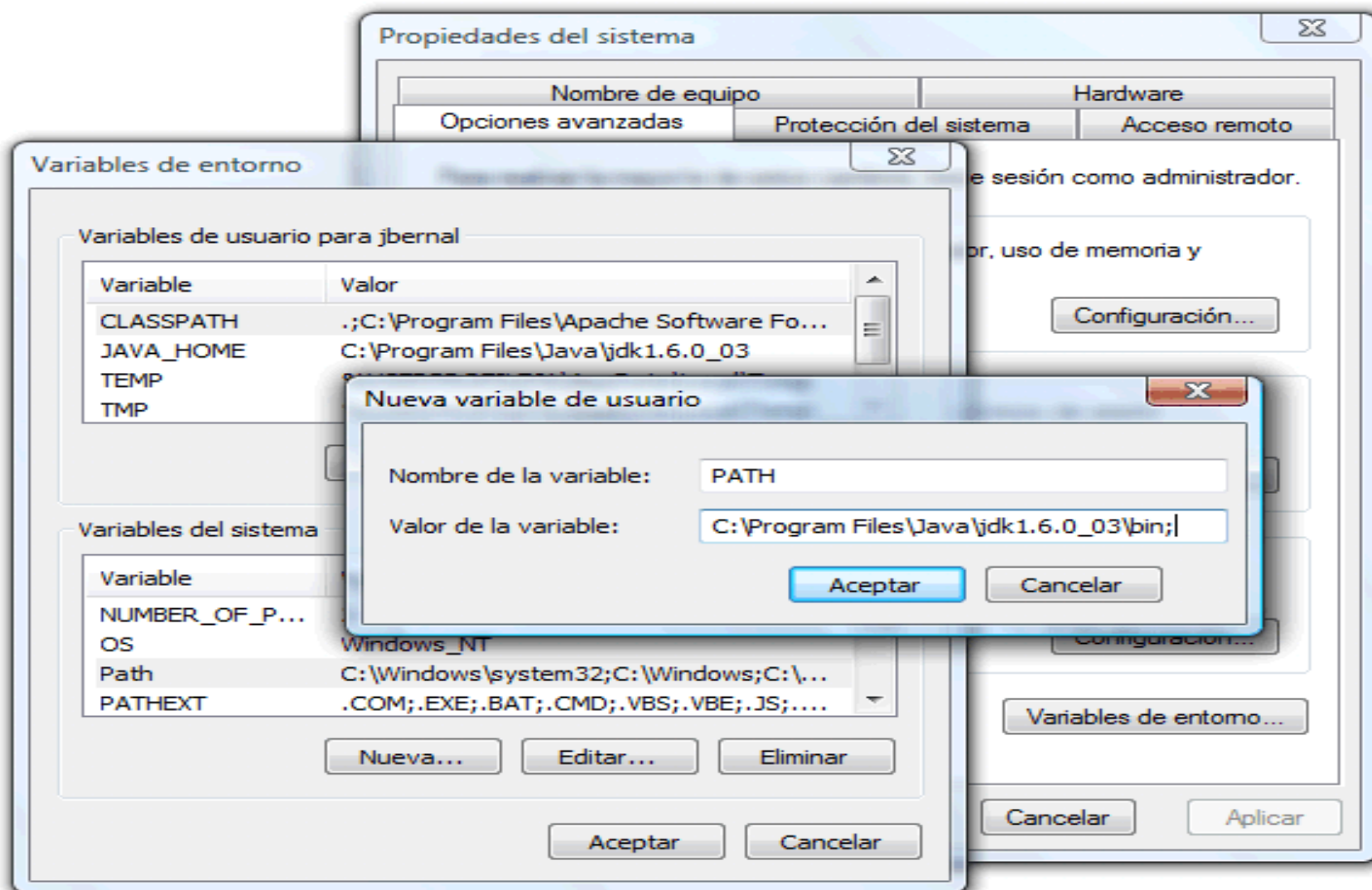
Java

- El lenguaje Java fue creado por Sun Microsystems Inc., Aparece en el año 1995
- Se creó en su origen para que fuese un lenguaje multiplataforma
 - Para ello se compila en un código intermedio: bytecode y necesita de una máquina virtual que lo ejecute. Normalmente, no utiliza código nativo, es decir, no se puede ejecutar directamente por el procesador
- Código fuente: *.java
- Código intermedio: *.class
- Tutorial:
 - <http://docs.oracle.com/javase/tutorial/>
- API:
 - <http://docs.oracle.com/javase/8/docs/api/index.html>

Entorno de desarrollo

- Para el desarrollo y compilación de aplicaciones Java, utilizaremos: Standard Edition (Java SE) o Java Development Kit (JDK) de Sun:
 - <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- Se trabaja mediante comandos en consola. Incluye, entre otras, las siguientes utilidades:
 - El compilador: **javac.exe**
 - El intérprete: **java.exe**
 - Un compresor: **jar.exe**
 - El generador de documentación: **javadoc.exe**
 - Un analizador de clases: **javap.exe**
- Para la ejecución de aplicaciones Java (Delivery Platforms) es el Java Runtime Environment (JRE). Se instala automáticamente con Java SE

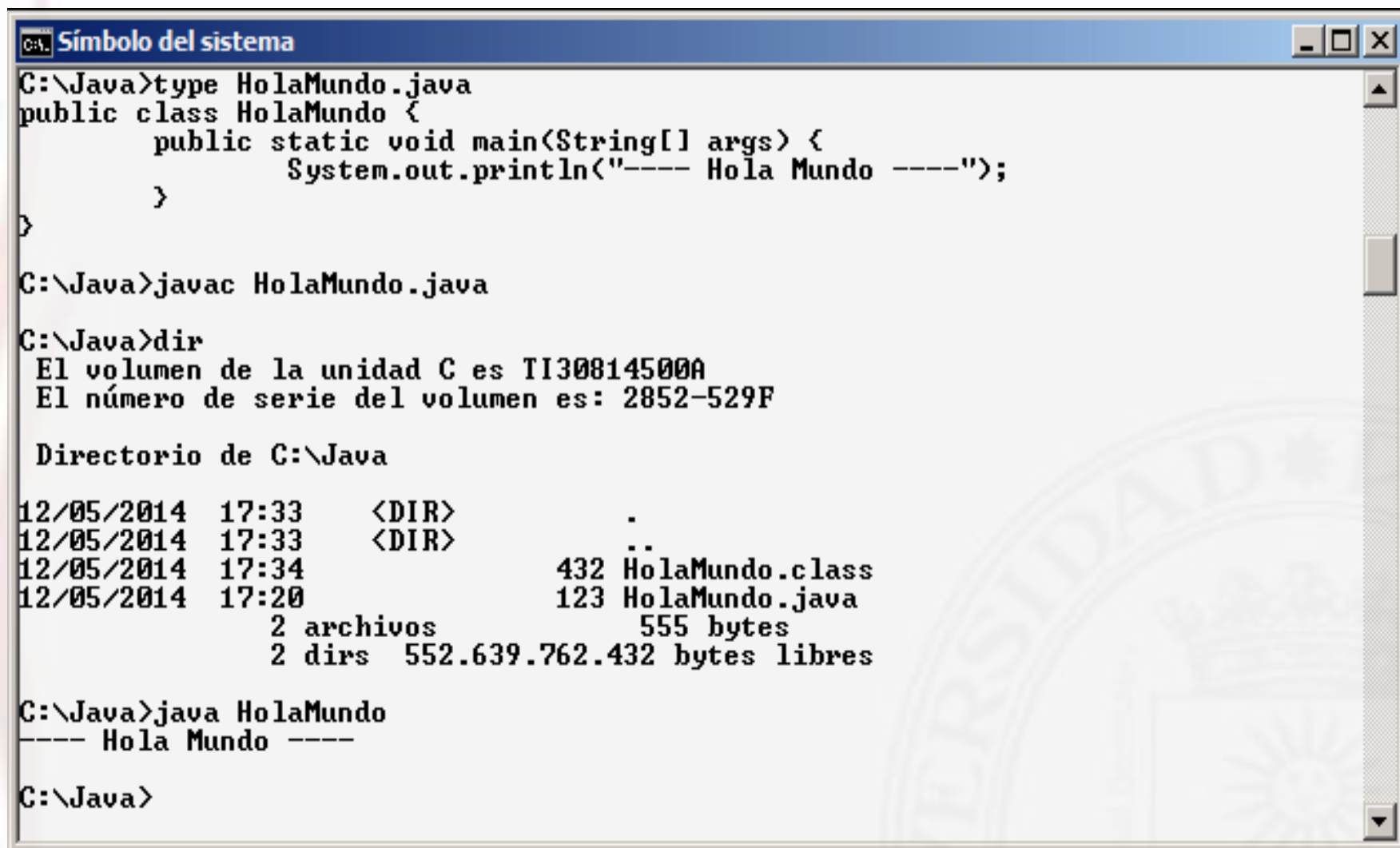
Entorno de desarrollo



Consola

- La consola es una ventana (llamada Símbolo del Sistema en Windows) que nos proporciona un punto de entrada para escribir comandos. Estos comandos nos permiten realizar tareas sin utilizar un entorno gráfico
- Aunque resulte un tanto incómodo, es una buena alternativa para la primera aproximación a Java.
- Cuando se inicia la consola, aparece la ruta de una carpeta y acaba con el símbolo ">". Cualquier comando Java lo deberemos ejecutar sobre la carpeta de trabajo
- Comandos básicos:
 - Para cambiar de unidad, escribiremos la letra de unidad seguida de dos puntos, por ejemplo: "c:"
 - **cd.** Con este comando cambiaremos de carpeta. Para subir un nivel escribiremos: "cd..", y para profundizar "cd carpeta"
 - **dir.** Lista el contenido de la carpeta
 - **type.** Visualiza el contenido de un fichero de texto: "type fich.java"
 - ↑,↓. Repite comandos anteriores

Consola



```
C:\Java>type HolaMundo.java
public class HolaMundo {
    public static void main(String[] args) {
        System.out.println("---- Hola Mundo ----");
    }
}

C:\Java>javac HolaMundo.java

C:\Java>dir
El volumen de la unidad C es TI30814500A
El número de serie del volumen es: 2852-529F

Directorio de C:\Java

12/05/2014  17:33    <DIR>          .
12/05/2014  17:33    <DIR>          ..
12/05/2014  17:34                432 HolaMundo.class
12/05/2014  17:20                123 HolaMundo.java
                2 archivos                555 bytes
                2 dirs  552.639.762.432 bytes libres

C:\Java>java HolaMundo
---- Hola Mundo ----

C:\Java>
```

IDE: Eclipse

- <http://www.eclipse.org/downloads/>
 - Eclipse IDE for Java Developers (Kepler Sr2)
- Eclipse es un entorno de desarrollo integrado (IDE) de código abierto multiplataforma para desarrollar todo tipo de aplicaciones. Concretamente se utiliza Eclipse para el desarrollo de aplicaciones Java.
- Además de los módulos iniciales, eclipse posee una serie de extensiones mediante plugins para integración con tecnologías que no provee en la distribución base
- Primeros pasos con Eclipse
 - Menu: > *File* > *New* > *Java Project...* **POO**
 - src: > *New* > *Package...* **basic**
 - package: > *New* > *Class...* **HolaMundo**
 - HolaMundo: > *Run As* > *Java application...*

Despliegue de aplicaciones: JAR

- Los ficheros Jar (Java ARchives) permiten recopilar en un sólo fichero varios ficheros diferentes, almacenándolos en un formato comprimido. Además, para utilizar todas las clases que contiene el fichero Jar no hace falta descomprimirlo
- El fichero Jar puede ser ejecutable, para ello se debe añadir un fichero manifiesto donde indicamos la clase principal (con método main)
- Eclipse
 - Proyecto Java: > *Export* > *Java/JAR file*
 - Proyecto Java: > *Export* > *Java/Runnable JAR file*
- Para ejecutar un fichero Jar:
 - Consola: > *java -jar ejecutable.jar*
 - En entorno gráfico: *doble click*

Clase IO

- La clase IO ha sido creada para facilitar la Entrada/Salida de datos en modo gráfico. Es una clase desarrollada propia y es independiente de la distribución Java SE. Se puede localizar en la ruta:
 - <http://www.eui.upm.es/~jbernal/io.jar>
- Para su instalación:
 - Copiar *io.jar*, en la carpeta *lib* del proyecto
 - *io.jar*: > *Build Path* > *Add to Build Path*
- Para su uso:
 - Añadir la sentencia *import* en la clase: **import** upm.jbb.IO;

Clase IO

- Métodos básicos:
 - Para leer un String: *IO.in.readString()* o *IO.in.readString("Mensaje")*
 - Para leer un int: *IO.in.readInt()* o *IO.in.readInt("Mensaje")*
 - Para leer un double: *IO.in.readDouble()* o *IO.in.readDouble("Mensaje")*
 - Para leer cualquier clase: *IO.in.read("paquete.Clase", "Mensaje")*
 - Para escribir un dato con salto de línea: *IO.out.println(...)*
 - Para escribir un dato sin salto de línea: *IO.out.print(...)*
 - Para escribir en la barra de estado: *IO.out.setStatusBar("mensaje")*

```
package basic;

import upm.jbb.IO;
public class HolaMundoIO {
    public static void main(String[] args) {
        IO.out.println("---- Hola Mundo ----");
    }
}
```


Clase IO

```
package basic;

import upm.jbb.IO;
public class PruebaIO {
    public static void main(String[] args) {
        String msg = IO.in.readString("Escribe un mensaje");
        IO.out.println("Valor del mensaje: " + msg);
        int entero = IO.in.readInt("Dame un número entero");
        IO.out.println("Valor del entero: " + entero);
        double decimal = IO.in.readDouble("Dame un numero decimal");
        IO.out.println("Valor del decimal: " + decimal);
        short corto = (short) IO.in.read("short", "Corto");
        IO.out.println("Valor del corto: " + corto);
        Integer entero2 = (Integer) IO.in.read(8, "Valor por defecto (Integer)");
        IO.out.println("Valor del entero: " + entero2);
        Byte b = (Byte) IO.in.read(new Byte("3"), "Valor por defecto (Byte)");
        IO.out.println("Valor del byte: " + b);
        Integer[] intArray = (Integer[]) IO.in.read("Integer[]", "Array de enteros");
        IO.out.print("Array de enteros: ");
        IO.out.println(intArray);
        IO.out.setStatusBar("Barra de estado");
        System.out.println("Este mensaje sale por la consola...");
    }
}
```

Gramática Java. Comentarios e identificadores

- Comentarios
 - Comentarios de línea: `//`, es comentario hasta final de línea
 - Comentarios de bloque: `/*` ... `*/`
 - Comentarios de documentación: `/**` ... `*/`. Se utiliza para documentar los aspectos públicos de las clases; mediante la herramienta javadoc.exe se genera documentación en formato HTML con el contenido.
- Los blancos, tabuladores y saltos de línea, no afectan al código
- Las mayúsculas y minúsculas son diferentes
- Los identificadores se forman mediante: {letra} + [letras | números | `_` | `$`].
- Existe un estilo de nombrar los diferentes elementos ampliamente aceptado por la comunidad Java:
 - Clases: las palabras empiezan por mayúsculas y el resto en minúsculas (ej. HolaMundo)
 - Métodos: las palabras empiezan por mayúsculas y el resto en minúsculas, excepto la primera palabra que empieza por minúsculas (ej. holaMundo)
 - Constantes: todo en mayúsculas separando las palabras con `_`
 - Paquetes: empiezan por minúsculas. Se jerarquizan con `.`

Clases: Atributos

- Los atributos son privados (*private*), y por lo tanto, no son directamente accesibles. La consulta o modificación de los atributos se debe realizar a través de métodos públicos; de esta manera, el objeto controla las operaciones válidas que se pueden realizar
- Un atributo es una información que se almacena y es parte representativa del estado. Los atributos se pueden sustentar mediante tipos primitivos o clases. Mostremos algunos ejemplos.
- La clase *Punto* representa un punto en un espacio de dos dimensiones, sus atributos son x e y
- La clase *NumeroNatural* representa un valor natural, su atributo podría ser *valor*

Clases: Atributos

```
package basic;
```

```
public class Punto {  
    private int x, y;  
}
```

```
package basic;
```

```
public class NumeroNatural {  
    private int natural;  
}
```

```
package basic;
```

```
public class Usuario{  
    private String nombre, eMail, ciudad;  
    private long movil;  
    private boolean mayorEdad;  
}
```

Gramática Java. Variables y constantes

- Existen dos tipos de variables: las *primitivas* y las de *clase*
 - Las variables primitivas almacenan el dato en sí, por ello, dependiendo del tipo de dato ocupan un tamaño diferente de memoria
 - Las variables de clase almacenan la referencia del objeto y tienen un tamaño fijo

- La declaración de variables tiene el siguiente formato:

```
[private] tipo identificador;  
[private] tipo identificador = expresión;  
[private] tipo identificador1, identificador2 = expresión;  
[private] final tipo IDENTIFICADOR = expresión;
```

```
[private] int prueba;  
[private] byte pruebaDos, prueba3 = 10, prueba4;  
[private] final int CONSTANTE_UNO = 10;
```

Gramática Java. Variables y constantes

- Enteros
 - *byte*. 8 bits. -128 a +127
 - *short*. 16 bits. -32.768 a +32.767
 - *int*. 32 bits. -2.147.483.648 a -2.147.483.648
 - *long*. 64 bits. -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
- Decimales
 - *float*. 32 bits. $\approx \pm 3.4 \cdot 10^{38}$
 - *double*. 64 bits. $\approx \pm 1.8 \cdot 10^{308}$
- Caracteres
 - *char*. 16. '\u0000' a '\uffff', 'a'
- Lógico
 - *boolean*. true y false

Gramática Java. Variables y constantes

- **Literales.** Los valores numéricos están en notación decimal
 - Para realizarlos en notación hexadecimal se añade “0x”, por ejemplo 0xffff
 - Para notación binaria “0b”
 - El tamaño por defecto de un entero es *int*, y de un decimal *double*. Para cambiar el tamaño a long se añade “L” al final (ej. 23L), para cambiar a float se añade ‘F’.
- Los caracteres van entre comillas simples (ej. ‘x’). Los caracteres escape van precedidos por ‘\’, algunos ejemplos son:
 - ‘\n’: retorno de carro
 - ‘\t’: tabulador
 - ‘\’’: comillas simples
 - ‘\”’: comillas dobles
 - ‘\\’: propia barra
 - ‘\b’: pitido
 - ‘\u----’: valor Unicode
- Los caracteres Unicode van precedidos por el carácter ‘u’ y en notación hexadecimal, algunos ejemplos serían: ‘\u0064’, ‘\u1234’.

Clase String

- Clase String. Su utilidad es almacenar una cadena de caracteres. Un ejemplo de uso sería:
 - `[private] String nombre="Jorge", ciudad="Madrid";`
- Una cadena de caracteres no puede ser dividida mediante un salto de línea. Para cadenas largas, se dividen en varias y se concatenan con el operador “+”, pudiéndose definir en varias líneas

Clases: Atributos

- Clase Natural
- Clase Fraccion
- Clase Rectangulo
- Clase Intervalo
- Clase Angulo
- Clase Moto

Clases: Métodos

- Podemos organizar los métodos en tres tipos, teniendo en cuenta aspectos sintácticos y semánticos:
 - Constructores. Métodos que inicializan los atributos de la instancia
 - Métodos para accesos directo a los atributos (getters & setters). Aunque estos métodos son realmente generales, los distinguimos por separado por tener unas normas concretas de estilo. La razón de establecer los atributos en vista privada, es poder controlar la modificación del estado del objeto y no permitir que este evolucione hacia estados incoherentes
 - Métodos genéricos. Realizan acciones utilizando los atributos, ya sea para modificarlos o simplemente consultarlos

Clases: Métodos Genéricos

- Formato básico de un método:

[public/private] {void/tipo} mtdo (tipo p1, tipo p2){}

- Calificador es *public* o *private*: se establece el tipo de vista
- Tipo devuelto: *void* (vacío) se indica que no devuelve nada, o se establece el tipo devuelto
- Nombre del método
- Parámetros. Por parámetros se pasan valores externos al objeto que condicionan la acción a realizar. Existen dos tipos de parámetros:
 - Tipos primitivos (byte, short, int...). Estos se pasan por valor, es decir, se realiza una copia del contenido en la variable que sustenta el parámetro
 - Clases y arrays. Estos se pasan por referencia, es decir, se pasa la dirección en el parámetro

- Con la sentencia *return*, se finaliza el método. Si además queremos devolver un valor se después de la sentencia

Clases: Métodos Genéricos

```
package basic;

public class Punto {
    private int x, y;

    public double modulo() {
        return 0; //Se calcula su valor
    }

    public double fase() {
        return 0; //Se calcula su valor
    }
}
```

Clases: Métodos Getters & Setters

- A nivel gramatical, son métodos genéricos, pero semánticamente, tienen connotaciones especiales
- Cada atributo tiene dos posibles acciones: leer su valor o establecerlo. No tenemos por qué realizar ambas acciones, depende del diseño, de hecho, muchas clases no tiene estos métodos
- Existe una nomenclatura especial para el identificador del método:
 - Establecer el valor de un atributo: set + NombreAtributo (la primera letra del nombre del atributo debe estar en mayúsculas). Acepta un solo parámetro del tipo del atributo y el mismo nombre
 - Leer el valor del atributo: get + NombreAtributo o is + NombreAtributo si devuelve un tipo boolean. El tipo devuelto coincide con el atributo, no acepta parámetros

Clases: Métodos Genéricos

```
package basic;
public class Punto {
    private int x, y;

    public void setX(int x) {
    }
    public void setY(int y) {
    }
    public int getX() {
        return 0;
    }
    public int getY() {
        return 0;
    }

    public double modulo() {
        return 0;
    }
    public double fase() {
        return 0;
    }
}
```


Clases: Métodos Constructores

- Los constructores son métodos especiales que reúnen las tareas de inicialización de los atributos de un objeto; por lo tanto, el constructor establece el estado inicial
- No es obligatorio definir constructores, si no se realiza, existe un constructor por defecto sin parámetros
- La ejecución del constructor es implícita a la creación de una instancia
- La restricción sintáctica del constructor es que debe llamarse igual que la clase y no devuelve ningún tipo ni lleva la palabra *void*

```
package basic;  
public class Punto {  
    private int x, y;  
    public Punto() {  
    }  
}
```

Clases: Métodos Sobrecarga

- La sobrecarga es definir dos o más métodos con el mismo nombre, pero con parámetros diferentes por cantidad o tipo. El objetivo de la sobrecarga es reducir el número de identificadores distintos para una misma acción pero con matices que la diferencian. La sobrecarga se puede realizar tanto en métodos generales, como en constructores
- El tipo devuelto puede variar
- La sobrecarga es un polimorfismo estático, ya que es el compilador quien resuelve el conflicto del método a referenciar
- Si definimos un constructor con parámetros, el constructor sin parámetros deja de estar disponible

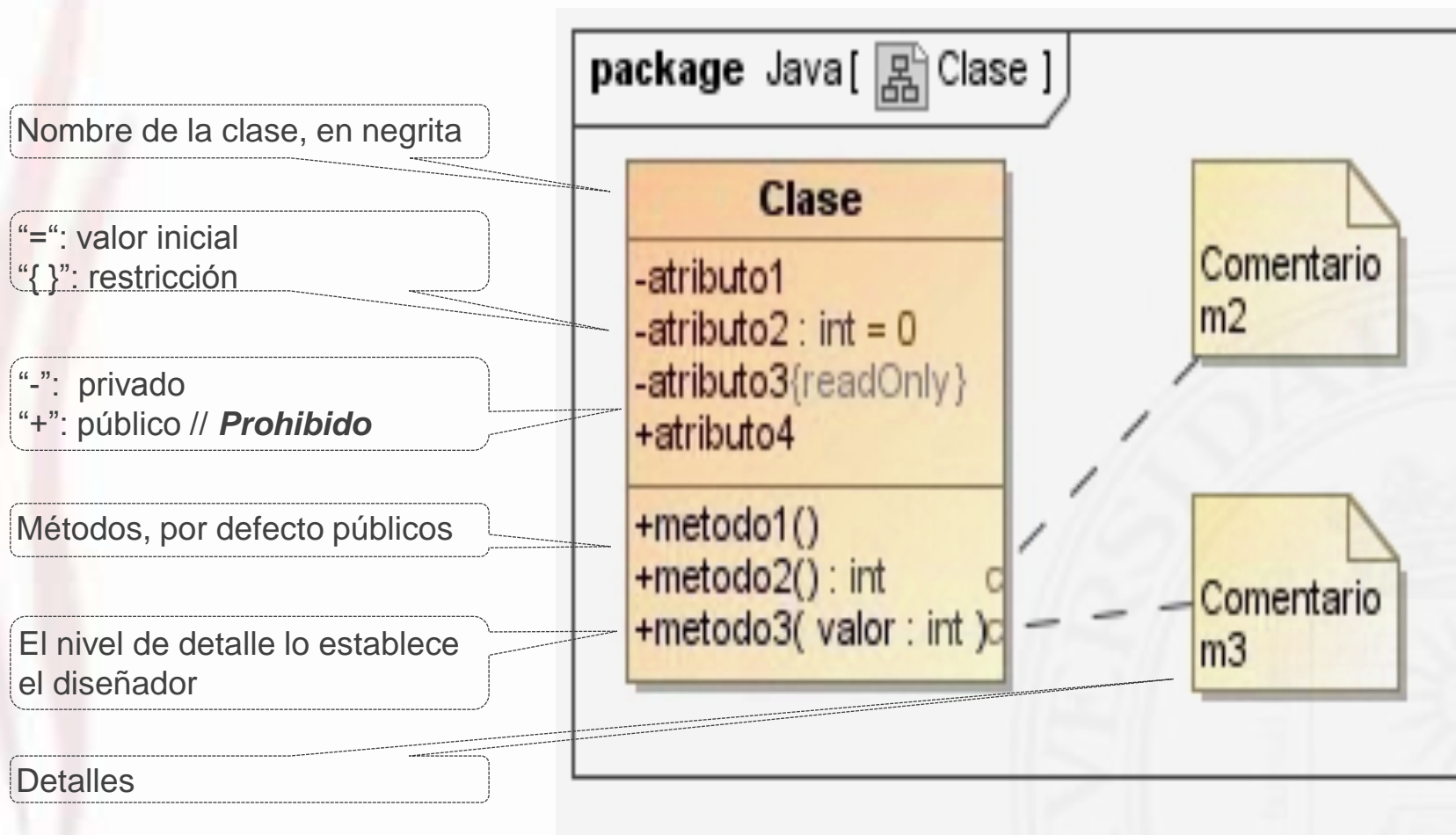
```
public class Punto {  
    private int x, y;  
    public Punto(int x, int y) {}  
    public Punto(int xy){}  
    public Punto() {}  
}
```

UML

- Es un lenguaje estándar, basado en una notación gráfica, que se utiliza para representar un modelo software
- Se representan diferentes vistas del modelo. Cada vista se concentra en un aspecto determinado
- Es una notación, no es una metodología, y por lo tanto, no dicta estándares para el proceso de desarrollo
- Se utiliza para especificar, visualizar, construir y documentar
- Se dispone de diferentes diagramas asociados al proceso de desarrollo. Utilizaremos aquellos diagramas relacionados con la fase que nos ocupa
- UML on-line:
 - <http://www.gliffy.com/>

Clases: UML

- Diagrama de clases
 - Proporcionan una perspectiva estática del código



Clase Punto

```
public class Punto {  
    private int x, y;  
  
    public Punto(int x, int y) {}  
    public Punto(int xy){}  
    public Punto() {}  
  
    public void setX(int x) {}  
    public void setY(int y) {}  
    public int getX() {  
        return 0;  
    }  
    public int getY() {  
        return 0;  
    }  
  
    public double modulo() {  
        return 0;  
    }  
    public double fase() {  
        return 0;  
    }  
}
```

Clases: Métodos

- Clase Natural
- Clase Fraccion
- Clase Rectangulo
- Clase Intervalo
- Clase Angulo
- Clase Moto

Operadores

- Asignación
 - `=`. Ej.: `x=y;`
- Aritméticos
 - `++`, `--`. Suma o resta 1, ej.: `x++`; `++x`;
 - `-`. Cambio de signo, ej.: `-x`;
 - `*`, `/`. Multiplicación y división, ej.: `x*y`;
 - `%`. Resto de la división, ej.: `x%y`;
 - `+`, `-`. Suma y resta, ej.: `x+2`;
 - `+=`, `-=`, `*=`, `/=`, `%=`. Ej.: `x+=2`; `x=x+2`;
- Binarios
 - `~`. Not, ej.: `~x`;
 - `<<`. Desplazamiento a la izquierda, se introducen ceros, ej.: `x<<2`;
 - `>>`. Desplazamiento a la derecha, se mantiene signo, ej.: `x>>2`;
 - `>>>`. Desplazamiento a la derecha introduciendo ceros
 - `&`, `|` y `^`. And, or y xor, ej.: `x&12`;
- Comparación
 - `==`, `!=`, `<`, `>`, `<=`, `>=`. Igual, distinto, menor que...
- Lógicos
 - `!`. Not lógico
 - `&&`, `&`, `||` y `^`. And, and perezoso, or, or perezoso y xor

Expresiones

- Las expresiones son una combinación de operadores y operandos. La precedencia y asociatividad determina de forma clara la evaluación. El orden aplicado es:
 - Paréntesis: ()
 - Unarios: + - !
 - Multiplicativos: * / %
 - Aditivos: + -
 - Relacionales: < > <= >=
 - Igualdad: == !=
 - Y lógico: & &&
 - O lógico: | ||

Arrays

- Los arrays son tablas de valores del mismo tipo. Son dinámicos y deben definirse y crearse. Son referencias
- Se define pero no se crea; el valor inicial es la palabra reservada *null*
 - `int[] tabla;`
- Se define y se crea. Los posibles valores de índice van desde 0 a tamaño-1:
 - `int[] tabla = new int[10];`
- Se define, se crea y se da contenido:
 - `int[] tabla = { 0, 1, 2 };`
- Para saber el tamaño de un array no nulo se puede utilizar el atributo *length*
 - `tabla.length`
- Se pueden definir arrays de varias dimensiones:
 - `int[][] matriz = new int[10][20];`
- Si un array lleva el calificador *final* no se puede volver a crear, pero si se permite cambiar el contenido del array
- Para leer o establecer el contenido de una array:
 - `valor = tabla[3] ;`
 - `tabla[3] = valor;`

Promoción y Casting

- Cuando en Java operamos con expresiones, los resultados intermedios se almacenan en variables temporales con el tipo de mayor capacidad que intervenga en la operación; excepto con *byte* y *short* que se promociona automáticamente a *int*.
- El *casting* es forzar la conversión de un tipo a otro. Se sitúa delante, y entre paréntesis, el nuevo tipo
 - `int i = 3;`
 - `byte b = (byte) i;`
 - `byte b2 = 2;`
 - `b2 = (byte) (b * 3);`
- Hay que tener cuidado que en las operaciones parciales no se pierda precisión
 - `int/int + double` // Error de precisión
- Soluciones:
 - `(double)int/int + double`
 - `1.0*int/int + double`

Sentencias

- En general, todas las sentencias acaban con “;”.
- Sentencia de bloques
 - Agrupa en una sentencia de bloque un conjunto de sentencias. Para ello se sitúan entre llaves

```
{ sentencia1; sentencia2; ...}
```

Sentencia *if*

- La sentencia *if* es una de las más básicas y nos permite controlar el flujo de control dependiendo de una condición. Los dos formatos posibles son:
 - `if (condicion) {sentencia1;}`
 - `if (condición) {sentencia1; } else {sentencia2;}`
- Sentencia *else-if*. Aunque la sentencia *else-if* no existe como tal, queremos presentarla por su forma especial de tabulación. Se considera una sentencia *else-if* si en la condición siempre se evalúa una misma variable o expresión

```
if (expresion) {  
    sentencia1;  
}  
else if (expresion2) {  
    sentencia2;  
}  
else if (expresion3) {  
    sentencia3;  
    sentencia4;  
}  
else {  
    // No se debiera llegar aquí  
}
```

Sentencia *switch*

- Esta sentencia evalúa una expresión, y dependiendo de su valor, ejecuta un conjunto de sentencias. Sólo se puede aplicar a valores primitivos, enumerados y a String. Un ejemplo es:

```
switch (Expresion) {  
    case valor:  
        sentencia1;  
        sentencia2;  
        break;  
    case valor2:  
    case valor3:  
        sentencia3;  
        break;  
    case valor4:  
        sentencia4;  
        break;  
    default: // No se debiera llegar aquí  
}
```

Sentencia *switch*

- Esta sentencia evalúa una expresión, y dependiendo de su valor, ejecuta un conjunto de sentencias. Sólo se puede aplicar a valores primitivos, enumerados y a String. Un ejemplo es:

```
switch (Expresion) {  
    case valor:  
        sentencia1;  
        sentencia2;  
        break;  
    case valor2:  
    case valor3:  
        sentencia3;  
        break;  
    case valor4:  
        sentencia4;  
        break;  
    default: // No se debiera llegar aquí  
}
```

Sentencia *while*

- Sentencia *while* y *do while*. Estas sentencias se utilizan para realizar bucles o repeticiones hasta que se cumpla una determinada condición, pero a priori es indefinido el número de veces que se puede repetir.
- La sentencia *while* tiene una repetición de 0 a n, y la sentencia *do while* de 1 a n. Un ejemplo es:

```
while (condicion) {  
    sentencia1;  
    sentencia2;  
    sentencia3;  
}  
  
do {  
    sentencia4;  
    sentencia4;  
} while (condicion)
```


Sentencia *for*

- Esta sentencia realiza un bucle mediante un índice de iteración y una condición de fin que depende del índice.
- Si los índices se declaran en el propio *for*, no se podrán utilizar fuera de éste.
- Sentencia *for each*. Esta sentencia es útil para procesar una colección de datos, donde sólo nos interesa el conjunto de datos y no el orden con que se procesan. Se puede aplicar a las clases *Collection* y a los *arrays*.

```
for (inicialización; terminación(mientras); operación) {  
    sentencia1;  
}  
  
for (int i = 1; i <= 10; i++) {}  
  
int i, j;  
for (i = 1, j = 1; (i <= 5) && (j <= 5); i++, j += 2) {}  
  
for (;;) {} // bucle infinito  
  
for (int item: array){  
    // en la variable item tenemos el dato  
}
```

Sentencias de bucle

- Sentencia *continue*
 - Con la sentencia *continue*, se continúa con la siguiente interacción del bucle; puede aparecer en varios lugares del bucle, pero normalmente asociado a una condición
- Sentencia *break*
 - Con la sentencia *break* abortamos el bucle, o en general, el bloque donde se ejecute

this

- Dentro de la implementación de un método, a veces se necesita referenciar a la propia instancia a la cual pertenece. Para ello está la palabra reservada *this*
 - Un ejemplo dentro de *Punto* sería: `this.x = 3`
- Resulta recomendable calificar con *this* a todas las referencias a los atributos y métodos de la propia clase; y resulta obligatorio cuando queramos distinguir entre un parámetro y una variable de instancia con el mismo nombre
- También referenciamos con la palabra *this* a nuestros propios constructores, por ejemplo `this(3,4)`

Clase Punto

```
public class Punto {  
    private int x, y;  
    public Punto(int x, int y) {  
        this.setX(x);  
        this.setY(y);  
    }  
    public Punto() {  
        this(0, 0);  
    }  
    public void setX(int x) {  
        this.x = x;  
    }  
    public void setY(int y) {  
        this.y = y;  
    }  
    public int getX() {  
        return this.x;  
    }  
    public int getY() {  
        return this.y;  
    }  
    public double modulo() {  
        return Math.sqrt(this.x * this.x + this.y * this.y);  
    }  
    public double fase() {  
        return Math.atan((double) this.y / this.x);  
    }  
    @Override public String toString() {  
        return "Punto[" + x + "," + y + "]";  
    }  
}
```

Clase NumeroNatural

```
public class NumeroNatural {
    private int natural;

    public NumeroNatural(int natural) {
        // this.natural = natural; //Error se salta el filtro
        this.setNatural(natural);
    }
    public NumeroNatural() {
        this(0);
    }

    public int getNatural() {
        return this.natural;
    }
    public void setNatural(int natural) {
        if (natural < 0) this.natural = 0;
        else this.natural = natural;
    }
    public void añadir(int valor) {
        // this.natural += valor; //Error se salta el filtro
        this.setNatural(this.natural + valor);
    }
}
```

Clases: implementación

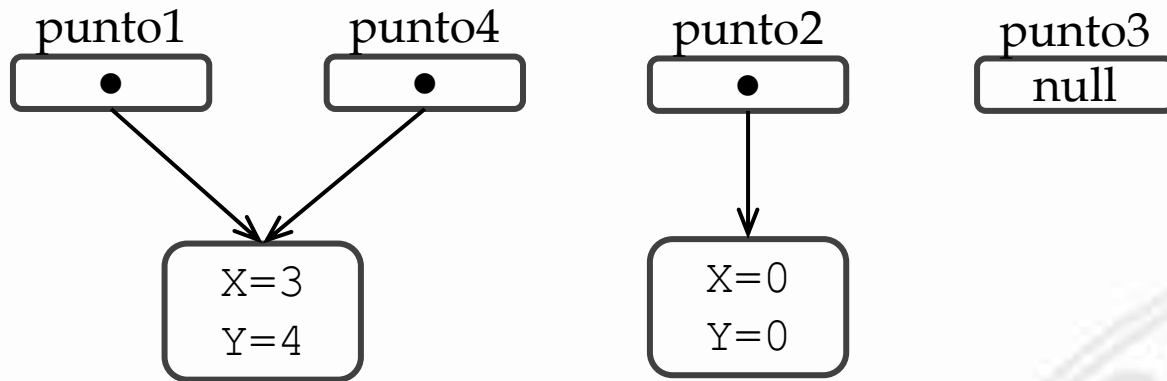
- Clase Natural
- Clase Fraccion
- Clase Rectangulo
- Clase Intervalo
- Clase Angulo
- Clase Moto
- Clase ContadorCircular
- Clase ColeccionDecimales
 - Método mayor (while)
 - Método menor (for)
 - Método media (for each)

Objetos

- Recordemos que la clase representa el modelo para crear objetos, pero son los objetos los elementos con los que podemos trabajar
- Los objetos ocupan memoria y tiene un estado, que representa el conjunto de valores de sus atributos
- Podemos crear todos los objetos que queramos, todos ellos tiene existencia propia, pudiendo evolucionar por caminos diferentes e independientes
- Cuando se crea un objeto se debe asociar con una referencia, por lo tanto, el proceso debe tener dos partes:
 - Declarar una variable que pueda referenciar al objeto en cuestión
 - Crear el objeto y asociarlo a la variable. Para crear el objeto, se realiza con la sentencia *new*, recordar que lleva implícito llamar al método constructor para que inicialice el objeto

Objetos

- `Punto punto1, punto2 = new Punto(), punto3, punto4;`
- `punto1 = new Punto(3,4);`
- `punto4 = punto1;`



- En Java, no existe ninguna forma explícita para destruir un objeto, simplemente cuando un objeto se queda huérfano (no tiene ninguna referencia) se destruye
- Esta labor la realiza el recolector de basura (*garbage collector*) de forma automática

Mensajes

- Una vez creado los objetos es posible pasarles mensajes, o invocarles métodos, para solicitar una acción. El objeto establece a que método le corresponde el mensaje recibido.
- Para enviar un mensaje se realiza con la notación *punto*. Dependiendo si almacenamos el valor devuelto del método, tenemos dos posibilidades:
 - `referencia.metodo(param1,param2...);`
 - `referencia.metodo();`
 - `var = referencia.metodo(param1,param2...);`
 - `var = referencia.metodo();`

```
Punto punto1 = new Punto(2,5);
punto1.setX(4);
int x = punto1.getX();
double d = punto1.modulo();
punto1.setX(punto1.getX() + 1); // suma en una unidad el valor de x
```

Elementos estáticos

■ *ATRIBUTOS ESTÁTICOS*

- Para definirlos se antepone al tipo la palabra *static*;
- Estos atributos están compartidos por todos los objetos de la clase y son accesibles desde cada uno de ellos
- Para el acceso a un atributo, en lugar de utilizar *this*, se utiliza el nombre de la clase: *Clase.atributo*
- Las constantes asociadas a la clase suelen ser estáticas
 - **private static final int** MAX = 100;

■ *MÉTODOS ESTÁTICOS*

- Al igual que los atributos, se antepone la palabra *static*
- No puede utilizar en el código la palabra *this*
- Se permite definir un código estático para inicializar los atributos estáticos. Su ejecución se dispara una sola vez
- **static {}**
- Los métodos que no leen ni modifican los atributos de instancia, son candidatos a ser estáticos

Pruebas Unitarias

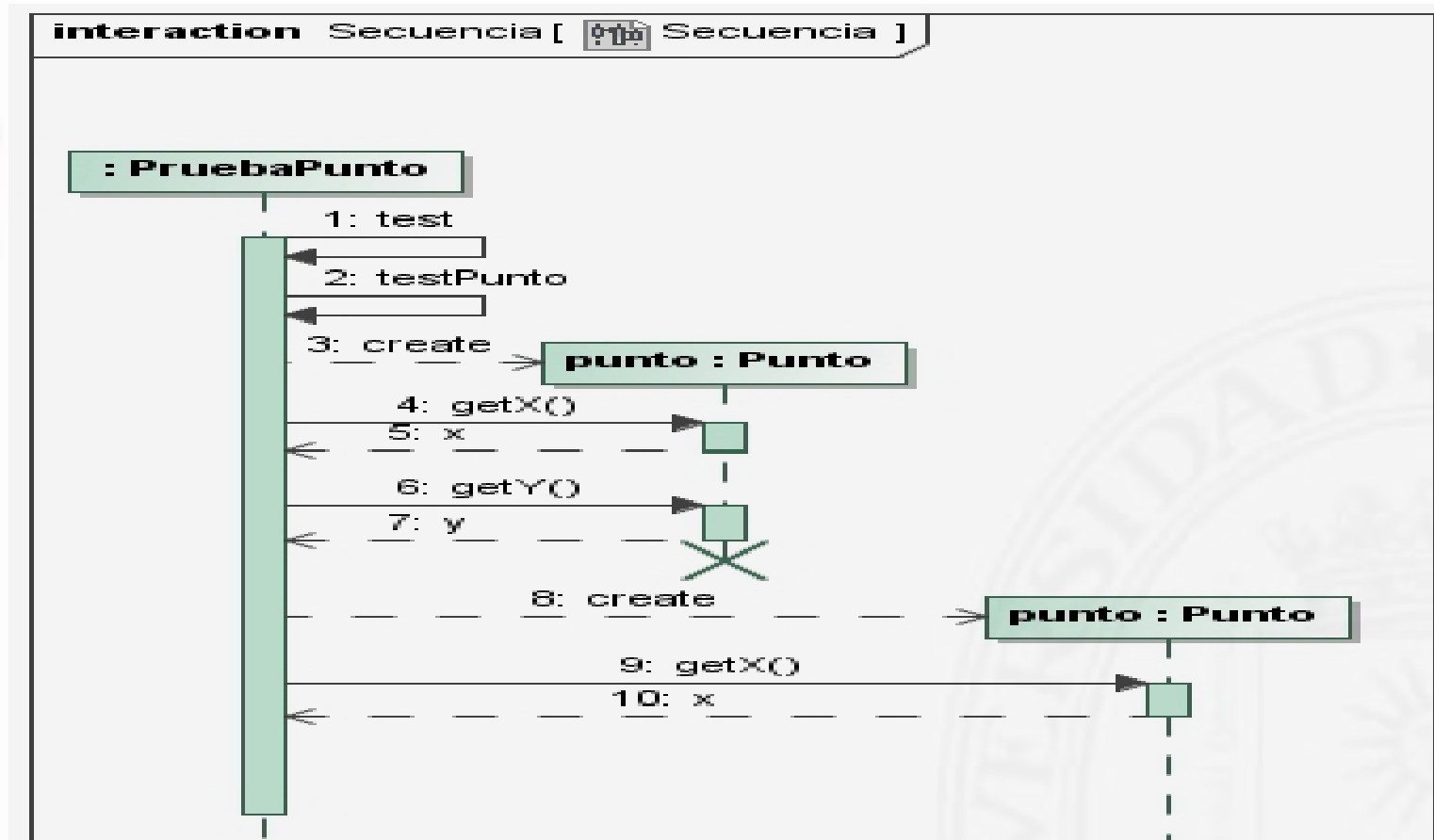
- Una prueba unitaria es una forma de probar el correcto funcionamiento de un módulo de código o clase independientemente del resto
- Característica de calidad:
 - Automática: clases que prueban clases
 - Cobertura: % de líneas de código ejecutadas
 - Repetibles: cuando parte del código ha sido modificado, se vuelven a lanzar las pruebas para comprobar que no se ha alterado su funcionalidad
 - Independiente: se prueban los módulos por separado

Pruebas Unitarias

```
public class PruebaPunto {
    public void testPunto() {
        Punto punto = new Punto();
        if (punto.getX() == 0 && punto.getY() == 0)
            System.out.println("Prueba constructor sin parámetros CORRECTA");
        else System.out.println("ERROR: p. constructor sin parámetros");
    }
    public void testPuntoIntInt() {
        Punto punto = new Punto(2, 2);
        if (punto.getX() == 2 && punto.getY() == 2)
            System.out.println("Prueba constructor con parámetros CORRECTA");
        else System.out.println("ERROR: p. constructor con parámetros");
    }
    public void testModulo() {
        Punto punto = new Punto(2, 3);
        if (punto.modulo() > 3.6055 && punto.modulo() < 3.6056)
            System.out.println("Prueba modulo CORRECTA");
        else
            System.out.println("ERROR: p. modulo:" + punto.toString() + ":" + punto.fase());
    }
    public void testFase() {
        Punto punto = new Punto(3, 3);
        if (punto.fase() > 0.7853 && punto.fase() < 0.7854)
            System.out.println("Prueba fase CORRECTA");
        else System.out.println("ERROR: p. fase:" + punto.toString() + ":" + punto.fase());
    }
    public void test() {
        this.testPunto();
        this.testPuntoIntInt();
        this.testModulo();
        this.testFase();
    }
    public static void main(final String[] args) {
        new PruebaPunto().test();
    }
}
```

UML: Diagrama de Secuencia

- Presenta en orden temporal la interacción de los objetos con los mensajes que se intercambian



Pruebas Unitarias: JUnit

- Característica de calidad:
 - Automática. Clases que prueban clases
 - Cobertura: % de líneas de código ejecutadas
 - Repetibles. Cuando parte del código ha sido modificado, se vuelven a lanzar las pruebas para comprobar que no se ha alterado su funcionalidad
 - Independiente. Se prueban los módulos por separado
- JUnit es un framework que nos ayuda a la realización de pruebas unitarias. Fue creado por Erich Gamma y Kent Beck. El framework incluye diferentes formas de ver los resultados (<http://www.junit.org/>)
- Test Case: Clases de prueba
- Test Suites: Contenedor de Test Case o Test Suites. Se crean estructura en árbol
- Asistente en Eclipse
 - Se enfoca la clase a testear > **new** > **JUnit Test Case**

JUnit. Ciclo de vida

- `@BeforeClass`: Se ejecuta una sola vez antes de la batería de pruebas definida en la clase y el método debe ser `static`
- `@Before`: Se ejecuta antes de cada uno de los marcados con `@Test` (es decir, si existen varios test, se ejecuta varias veces). Suele ser una inicialización por todas las pruebas de la clase
- `@Test`: Marca un método como prueba.
- `@Ignore`: Marca un método o una clase completa para que no se ejecute
- `@After`: Se ejecuta después de cada uno de los `@Test`. Suele ser una liberación de recursos
- `@AfterClass`: Se ejecuta al final del proceso completo y el método debe ser `static`


JUnit. Comprobaciones

- `assertEquals (valor_esperado, valor_real);`
 - Los valores pueden ser de cualquier tipo
 - Si son arrays, no se comprueban elemento a elemento, sólo la referencia
- `assertTrue (condición_booleana)`
- `assertFalse (condición_booleana)`
- `assertSame (Objeto esperado, Objeto real)`
 - Comprueba que son la misma referencia
- `assertNotSame (Objeto esperato, Objeto obtenido)`
 - Comprueba que son referencias distintas
- `assertNull (Objeto)`
 - Comprueba que el objeto es Null
- `assertNotNull (Objeto objeto)`
 - Comprueba que el objeto no es Null
- `fail (string Mensaje)`
 - Imprime el mensaje y falla
 - Útil para comprobar que se capturan excepciones

Pruebas Unitarias: PuntoTest

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class PuntoTest {
    @Test public void testPuntoIntInt() {
        Punto pt = new Punto(2, 2);
        assertEquals(pt.getX(),2);
        assertEquals(pt.getY(),2);
    }
    @Test public void testPunto() {
        Punto pt = new Punto();
        assertEquals(pt.getX(),0);
        assertEquals(pt.getY(),0);
    }
    @Test public void testModulo() {
        Punto pt = new Punto(2,3);
        assertEquals(pt.modulo(),3.6055,0.0001);
    }
    @Test public void testFase() {
        Punto pt = new Punto(3,3);
        assertEquals(pt.fase(),0.7853,0.0001);
    }
}
```

Pruebas Unitarias: PuntoTest

package Java[ PuntoTest]

Punto

-x : int

-y : int

+Punto(x : int, y : int)

+Punto()

+getX()

+getY()

+setX()

+setY()

+modulo() : double

+fase() : double

x, y, modulo, fase

2, 3, 2.2360, 0.4636

3, 3, 4.2426, 0.7854


-4, 2, 4.4721, -0.4636

1, 0, 1, 0

Pruebas Unitarias: PuntoTest

```
public class PuntoTest {
    private int[] xs = {2, 3, -4, 1}; private int[] ys = {1, 3, 2, 0};
    private double[] modulus = {2.2360, 4.2426, 4.4721, 1};
    private double[] fases = {0.4636, 0.7854, -0.4636, 0};
    private void testPuntoIntInt(int x, int y) {
        Punto pt = new Punto(x, y);
        assertEquals(pt.getX(), x); assertEquals(pt.getY(), y);
    }
    @Test public void testPuntoIntInt() {
        for (int i = 0; i < xs.length; i++) { this.testPuntoIntInt(xs[i], ys[i]); }
    }
    @Test public void testPunto() {
        Punto pt = new Punto();
        assertEquals(pt.getX(), 0);
        assertEquals(pt.getY(), 0);
    }
    private void testModulo(int x, int y, double modulo) {
        Punto pt = new Punto(x, y);
        assertEquals(pt.modulo(), modulo, 0.0001);
    }
    @Test public void testModulo() {
        for (int i = 0; i < xs.length; i++) { this.testModulo(xs[i], ys[i], modulus[i]); }
    }
    private void testFase(int x, int y, double fase) {
        Punto pt = new Punto(x, y);
        assertEquals(pt.fase(), fase, 0.0001);
    }
    @Test public void testFase() {
        for (int i = 0; i < xs.length; i++) { this.testFase(xs[i], ys[i], fases[i]); }
    }
}
```

Clases: Test

package Java[ Test]

Rectangulo

-base : int{">=0}"
-altura : int{">=0}"

+area() : double
+perimetro() : double

base,altura,area,perimetro
1,1,1,4
0,1,0,2
0,0,0,0
-1,1,0,2 => 0,1
1,-1,0,2 =>1,0

Angulo

-grado : int{"0..360}"

+suma(grado : int)

grado,grado suma, grado
-,0 +10,10
-10,350 +10,0
10,10 -20,350
370,10 +730,20
730,10 -730,0
-370,350

Intervalo

-minimo : int{"<=maximo}"
-maximo : int{">=minimo}"

+solapado() : boolean
+incluido() : boolean

mínimo,máximo	otro	solapado	incluido
--=>0,0	2,2	false	false
-4,8=>-4,8	8,10	true	false
8,4=>8,8	8,8	true	true
2,4=>2,4	1,5	true	false
2,8=2,8	4,6	true	true

Clases Fundamentales

■ String

- Representa una cadena de caracteres. Su contenido no puede cambiar, por lo tanto, cada vez que se modifica, se crea un nuevo String
- Consume pocos recursos para el almacenamiento pero las operaciones de modificación resultan costosas en el tiempo. Si van a existir numerosas modificaciones se deberá utilizar la clase StringBuffer
- Métodos
 - Longitud de la cadena (*length*).
 - Comparar cadenas (*equals*, *equalsIgnoreCase*, *compareTo*).
 - Busca subcadenas (*indexOf*, *lastIndexOf*)
 - Extrae subcadenas (*substring*, *toLowerCase*, *toUpperCase*, *trim*)

■ StringBuffer

- Representa una cadena de caracteres con una estructura interna que facilita su modificación, ocupa más que la clase String, pero las operaciones de modificación son muchos más rápidas, pudiendo ser 1000 veces más rápida

Clases Fundamentales

- **CLASE NUMBER: INTEGER, DOUBLE...**
 - Este conjunto de clases son útiles para trabajar con los valores primitivos, pero mediante objetos. A estas clases se les llama *Wrappers* (envolventes). Cada clase especializada, almacena un tipo de valor primitivo numérico
 - Las clases tienen constantes que nos pueden ser útiles, por ejemplo, todas tienen MAX_VALUE y MIN_VALUE, *Double* además tiene NaN, POSITIVE_INFINITY. NEGATIVE_INFINITY...
 - Otra utilidad interesante es que pueden convertir una cadena a un valor numérico, por ejemplo, Integer.parseInt("123") devuelve el valor *int* asociado
 - Para facilitar su uso, en las versiones actuales de Java existe una conversión automática entre el tipo primitivo y el objeto (*boxing* y *unboxing*)
- **BOOLEAN, CHARACTER, MATH, SYSTEM...**

```
Integer claseI;  
Double claseD;  
int i;  
double d;  
claseI = 23;  
claseD = 2.2;  
i = claseI;  
d = claseD;
```


Herencia

- La herencia es uno de los mecanismos fundamentales de la programación orientada a objetos
 - Una clase se construye a partir de otra, buscando la mejora de las prestaciones de la clase heredada
 - Provee el polimorfismo, permite interactuar con la clase Padre, sin saber que instancia concreta de las clases Hija aporta la funcionalidad
- La herencia relaciona las clases de manera jerárquica
 - Una clase padre, superclase o clase base
 - Clases hijas, subclasses o clase derivada. Los descendientes de una clase heredan todos los atributos y métodos que sus ascendientes hayan especificado como heredables
- En Java, sólo se permite la herencia simple, es decir, la jerarquía de clases tiene estructura de árbol
- El punto más alto de la jerarquía es la clase *Object* de la cual derivan todas las demás clases
- Para especificar la superclase se realiza con la palabra *extends*; si no se especifica se hereda de *Object*.

```
public class Punto {} //se hereda de Object
public class Punto extends Object {} //es lo mismo que la anterior
public class PuntoTiempo extends Punto {}
```

Herencia

- Se heredan los atributos y métodos que sean *public*, *protected* o *friendly*. Los miembros que sean *private*, no se podrán invocar directamente mediante *this*, pero su funcionalidad esta latente en la herencia
- Los elementos heredados se referencian por *this* desde la implementación de la clase hija
- Para referirnos a los miembros de la clase padre, sólo si existe conflicto de nombres, se realizará mediante la palabra *super*
 - *super.miembro*: se accede a un atributo o método
 - *super()*: accedemos a los constructores
- La primera sentencia de un constructor es invocar un constructor de la clase padre; ,sino se especifica, se llama al constructor sin parámetros
- Calificador *final*
 - Si lo aplicamos a una clase, no se permite que existan subclases
 - Si lo aplicamos a un método, no se permite que subclases redefinan el método

Herencia: Clase PuntoTiempo

```
public class PuntoTiempo extends Punto {
    private int t;

    public PuntoTiempo(int x, int y, int t) {
        super(x, y); // Si no se especifica, es: super();
        this.setT(t);
    }
    public PuntoTiempo() {
        this(0, 0, 0);
    }

    public int getT() {
        return this.t;
    }
    public void setT(int t) {
        this.t = t;
    }

    public double velocidad() {
        return this.modulo() / this.getT();
    }
}
```

Compatibilidad entre variables de clases

- Una variable de tipo *MiClase*, puede contener instancias de *MiClase* o de cualquier *subclase*
- Una variable de tipo *Object*, puede contener cualquier instancia
- **No** está permitido que una variable tenga instancias de las superclases
- Una variable de tipo *Punto*, puede referencia instancias de tipo *PuntoTiempo*, pero una variable de tipo *PuntoTiempo* **NO** puede tener instancia de la clase *Punto*
- También se puede realizar casting entre clases, no recomendado

```
Punto p = new PuntoTiempo(1, 2, 3);  
// Error: PuntoTiempo pt = new Punto(3,2);  
p.setX(3);  
// Error con p.setT(3);  
((PuntoTiempo) p).setT(3); // Error con (PuntoTitmpo)p.setT(3);
```

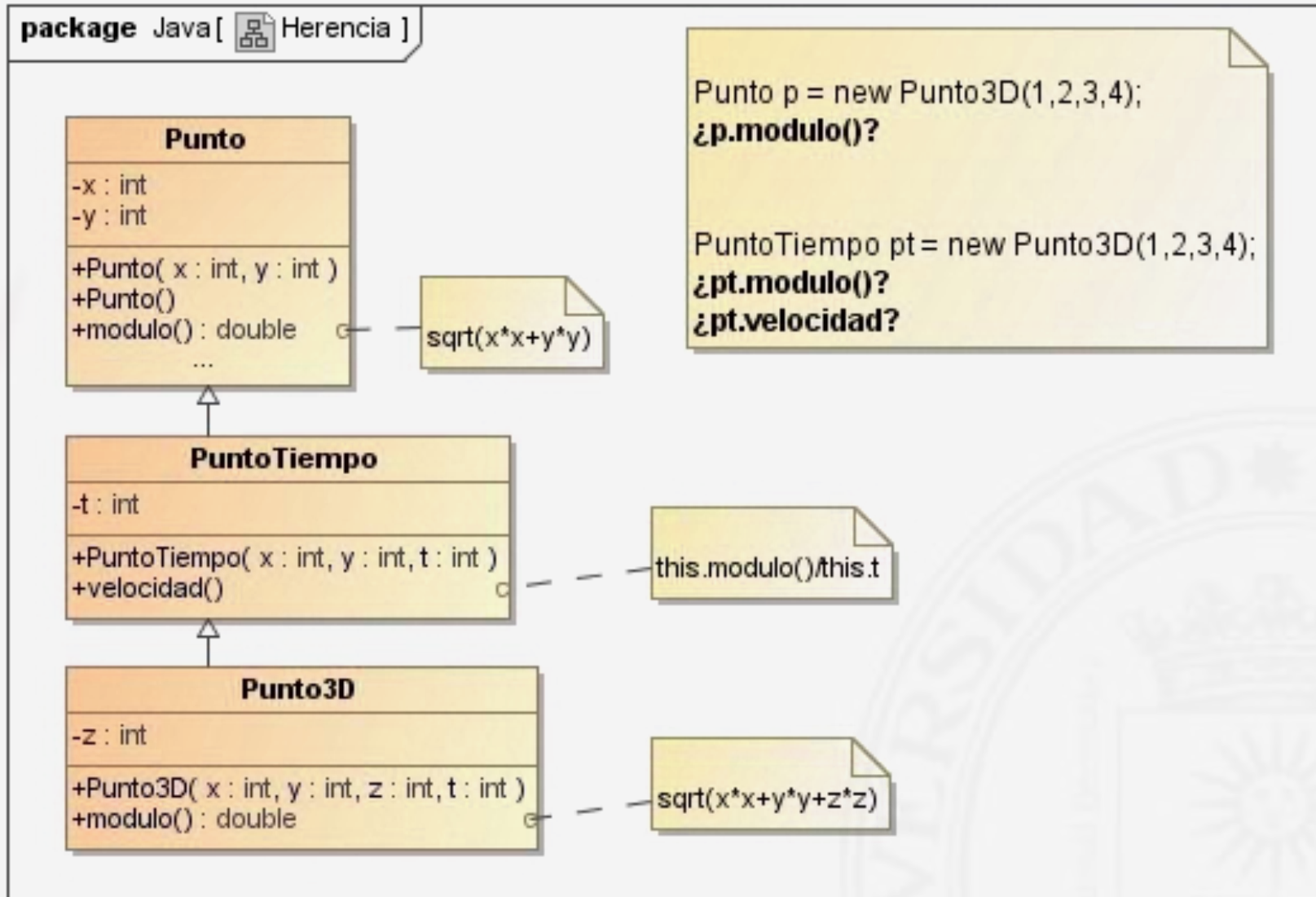
Redefinición de métodos

- La redefinición o sobrescritura consiste en que la clase hija cambia la implementación de un método heredado
- Se debe añadir la anotación de `@Override` para asegurarnos que no realizamos una sobrecarga por error
- Supongamos que queremos realizar una clase *Punto3D* que hereda de *PuntoTiempo* para añadirle una tercera dimensión. En este caso, el método heredado *modulo* ya no es válido y lo debemos redefinir

```
public class Punto3D extends PuntoTiempo {
    private int z;
    public Punto3D(int x, int y, int z, int t) {
        super(x, y, t);
        this.setZ(z);
    }
    public int getZ() { return this.z; }
    public final void setZ(int z) { this.z = z; }

    @Override public double modulo() {
        return Math.sqrt(this.getX() * this.getX() + this.getY() * this.getY()
            + this.z * this.z);
    }
}
```

Herencia



Clase Object

- Esta clase es la superclase de todas. Algunos de sus métodos son:
 - *public String toString()*. Este método devuelve una cadena con la descripción de la clase. Es importante redefinir este método para adaptarlo a nuestra descripción, es muy útil en los procesos de depuración
 - *public boolean equals(Object obj)*. Este método devuelve un boolean indicando si la instancia actual es igual a la que se pasa por parámetro. En *Object* compara direcciones, devolviendo true si son la misma dirección. Normalmente, esta no es la implementación necesitada
 - *public int hashCode()*. Devuelve un código hash de este objeto. Normalmente, si se sobrescribe *equals* se debe sobre escribir *hashCode*. Se debe buscar que el código hash sea diferente entre dos instancias que no se consideren iguales
 - *protected void finalize() throws Throwable*. Este es el método destructor, se debe redefinir si queremos realizar alguna acción para liberar recursos. Cuando un objeto queda huérfano, el recolector de basura lo destruye; pero esta acción no es inmediata. Si queremos que actúe el recolector, lo podemos lanzar mediante: *System.gc()*

Clase Object

```
@Override
public String toString() {
    return "Punto[" + x + "," + y + "]";
}

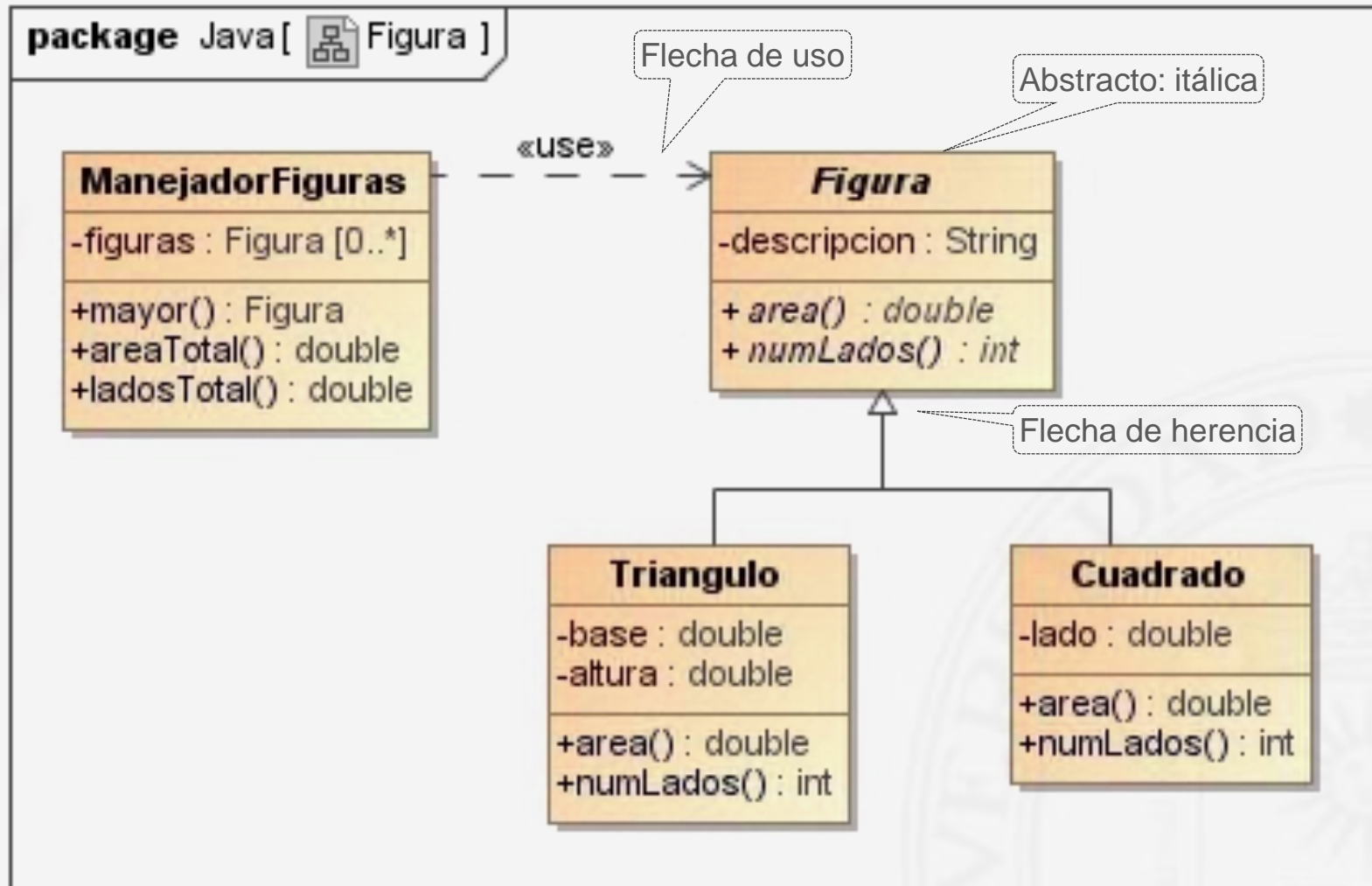
@Override
public boolean equals(Object obj) {
    boolean result;
    if (obj == null || this.getClass() != obj.getClass()) {
        result = false;
    } else {
        final Punto otro = (Punto) obj;
        result = otro.getX() == this.x && otro.getY() == this.y;
    }
    return result;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = prime + x;
    result = prime * result + y;
    return result;
}
```

Clases Abstractas

- Una clase abstracta es una clase que no se puede instanciar, pero si tiene constructores. Se debe añadir el calificador *abstract* después del calificativo *public*
- Se pueden definir métodos sin implementar y obligar a las subclasses a implementarlos
- Ejemplo. Tenemos la clase *Figura*, que representa una figura general, y subclasses con figuras concretas (*Triangulo*, *Circulo*...). Podemos establecer métodos comunes como *area*, que sin conocer el tipo de figura, sea capaz de calcularla. Pero para que esto funcione correctamente, cada figura concreta debe implementar su versión particular de *area*. La ventaja es que podemos trabajar con variables o parámetros de tipo *Figura* y llamar a los métodos comunes sin saber a priori el tipo concreto de *Figura*. Esto permite en el futuro, añadir nuevas *Figuras*, sin cambiar las clases ya desarrolladas. A este concepto se le llama polimorfismo
- El polimorfismo es el modo en que la POO implementa la polisemia, es decir, un solo nombre para muchos conceptos
- Este tipo de polimorfismo se debe resolver en tiempo de ejecución y por ello hablamos de polimorfismo dinámico

Clases Abstractas



Interfaces

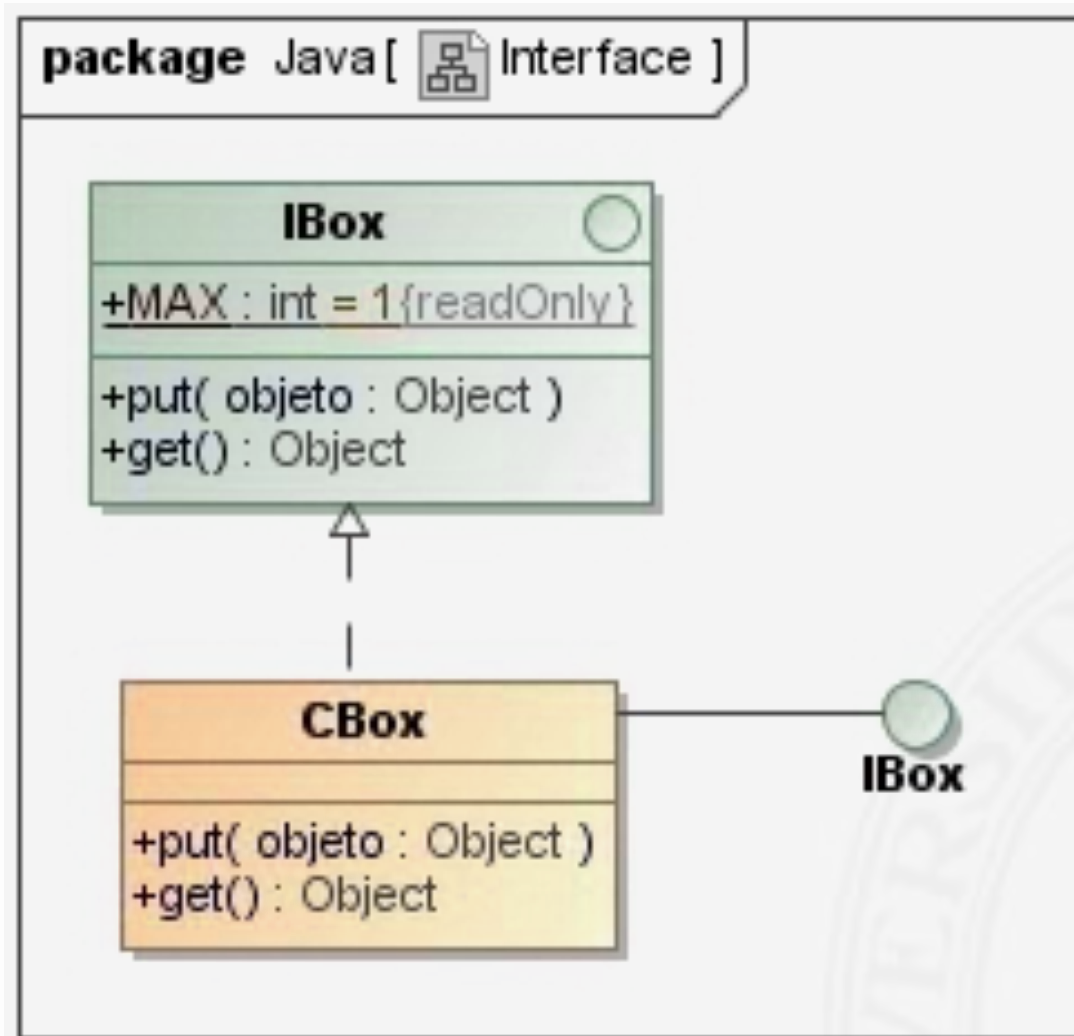
- Un interface es un tipo abstracto que representa un conjunto de métodos que las clases pueden implementar y un conjunto de constantes estáticas, es similar a una clase abstracta pura
- Los interfaces no se pueden instanciar
- Su utilidad es crear variables de tipo interface e independizarse de las clases que contiene
- Para definir un interface se realiza de forma similar a una clase pero con la palabra *interface*
- Pueden existir variables o parámetros de tipo interface, pero sólo pueden contener instancias de clases que hayan implementado el interface
- Una clase puede implementar uno o varios interfaces, para ello se utiliza la palabra *implements*

Interfaces

```
public interface IBox {  
    public static final int MAX = 1;  
    public void put(Object objeto);  
    public Object get();  
}
```

```
public class CBox implements IBox {  
    private Object objeto;  
    public CBox(){  
        this.objeto = null;  
    }  
    @Override  
    public Object get() {  
        return this.objeto;  
    }  
    @Override  
    public void put(Object objeto) {  
        this.objeto = objeto;  
    }  
}
```

Interfaces: UML



Interfaces

- Los interfaces pueden heredar de otros interfaces mediante la palabra *extends*, además se permite herencia múltiple
- Mediante interfaces también se aporta el polimorfismo, de hecho, una clase abstracta con todos sus métodos abstractos es similar a un interface
- Clases Abstractas vs Interfaces
 - Clases abstractas cuando diseñamos una jerarquía de clases relacionadas para organizar una temática
 - Interfaces cuando es un contrato entre clases para evitar su acoplamiento
- Compatibilidad de variables
 - Una variable de tipo interface puede contener instancias de cualquier clase que implemente el interface
 - `IBox var = new CBox();`

Colaboración entre clases: relaciones

- Una Aplicación OO es una colección de objetos con un único método main
- Cohesión y Acoplamiento (dependencias) entre clases
- Herencia
- Asociación
 - Uso
 - Agregación
 - Composición
- Multiplicidad: 1, 0..1; 0..*, 1..*

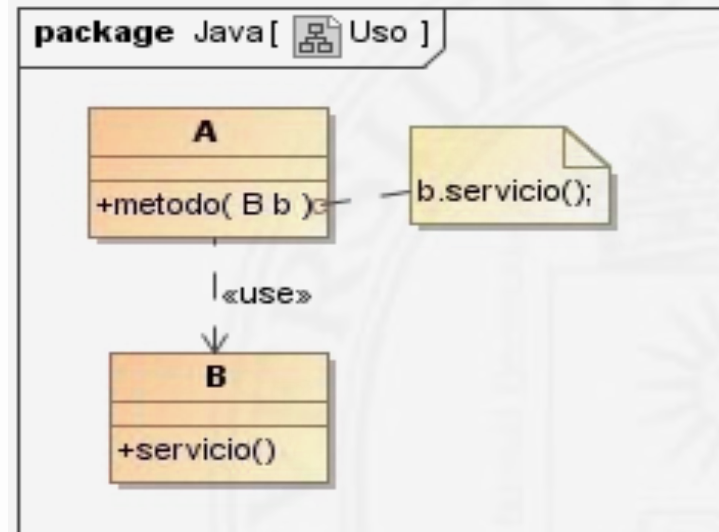
```
public class Una{  
    private Otra otra; // 0..1 o 1
```

```
public class Una{  
    private Otra[] otra; //0..* o 1..*
```

Relación de Uso

- Se dice que existe una relación de uso entre la clase A y la clase B, si un objeto de la clase A lanza mensajes al objeto de la clase B, para utilizar sus servicios
- Es una relación que se establece entre un cliente y un servidor, en un instante dado, a través de algún método
- No es una relación duradera

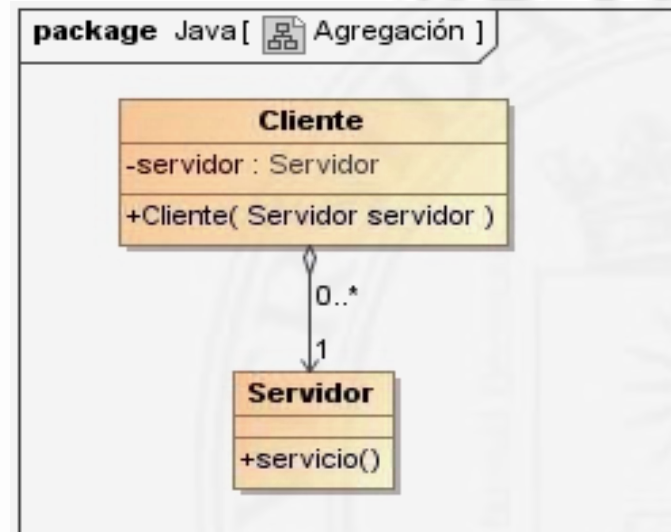
```
public class A {  
    //...  
    public void metodo(B b){  
        b.servicio();  
        //...  
    }  
    //...  
}
```



Relación de Agregación

- Se dice que la clase A tiene una relación de agregación con la clase B, si un objeto de la clase A delega funcionalidad en los objetos de la clase B
- El ciclo de vida de ambos no coincide
- La clase A delega parte de su funcionalidad, pero no crea el objeto B. Varios objetos pueden estar asociados con el objeto B
- Al objeto A, le deben pasar la referencia del objeto B
- Es una relación duradera entre cliente (A) y servidor (B), en la vida del cliente

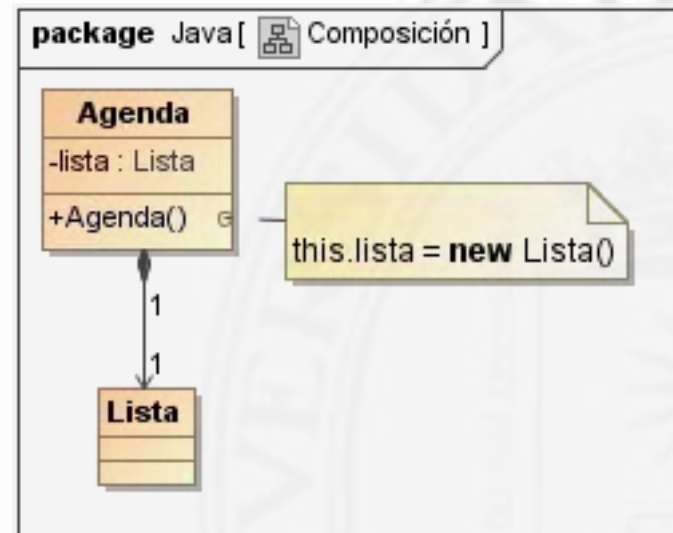
```
public class Cliente {  
    private Servidor servidor;  
  
    public Cliente(Servidor servidor) {  
        this.servidor = servidor;  
    }  
    //...  
}
```



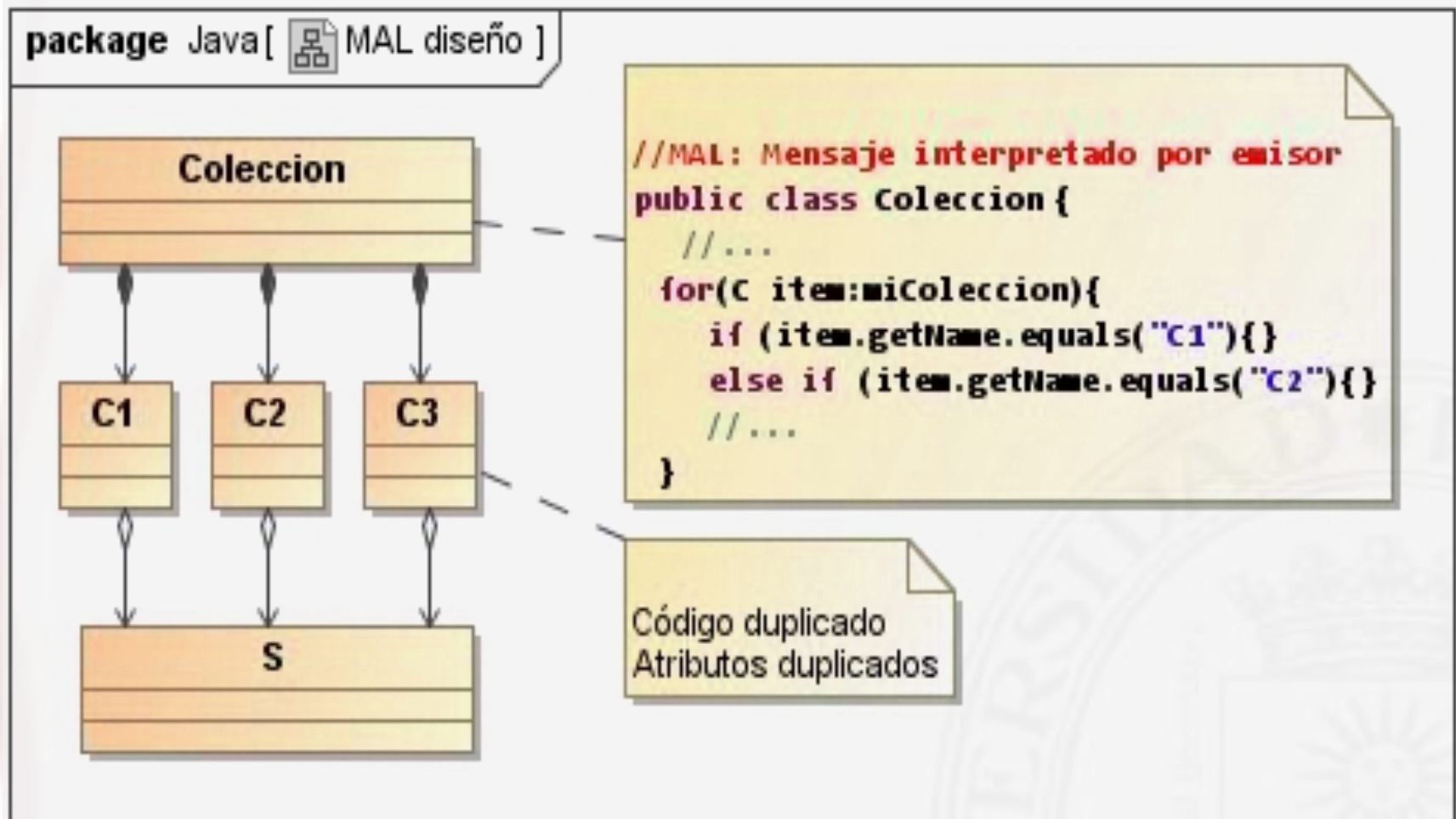
Relación de Composición

- Se dice que existe una relación de composición entre la clase A y la clase B, si la clase A contiene un objeto de la clase B, y delega parte de sus funciones en la clase B
- Los ciclos de vida de los objetos de A y B coinciden
- El objeto A es el encargado de crear la instancia de B y de almacenarla, siendo su visibilidad privada
- Cuando se destruye A, también se destruye B

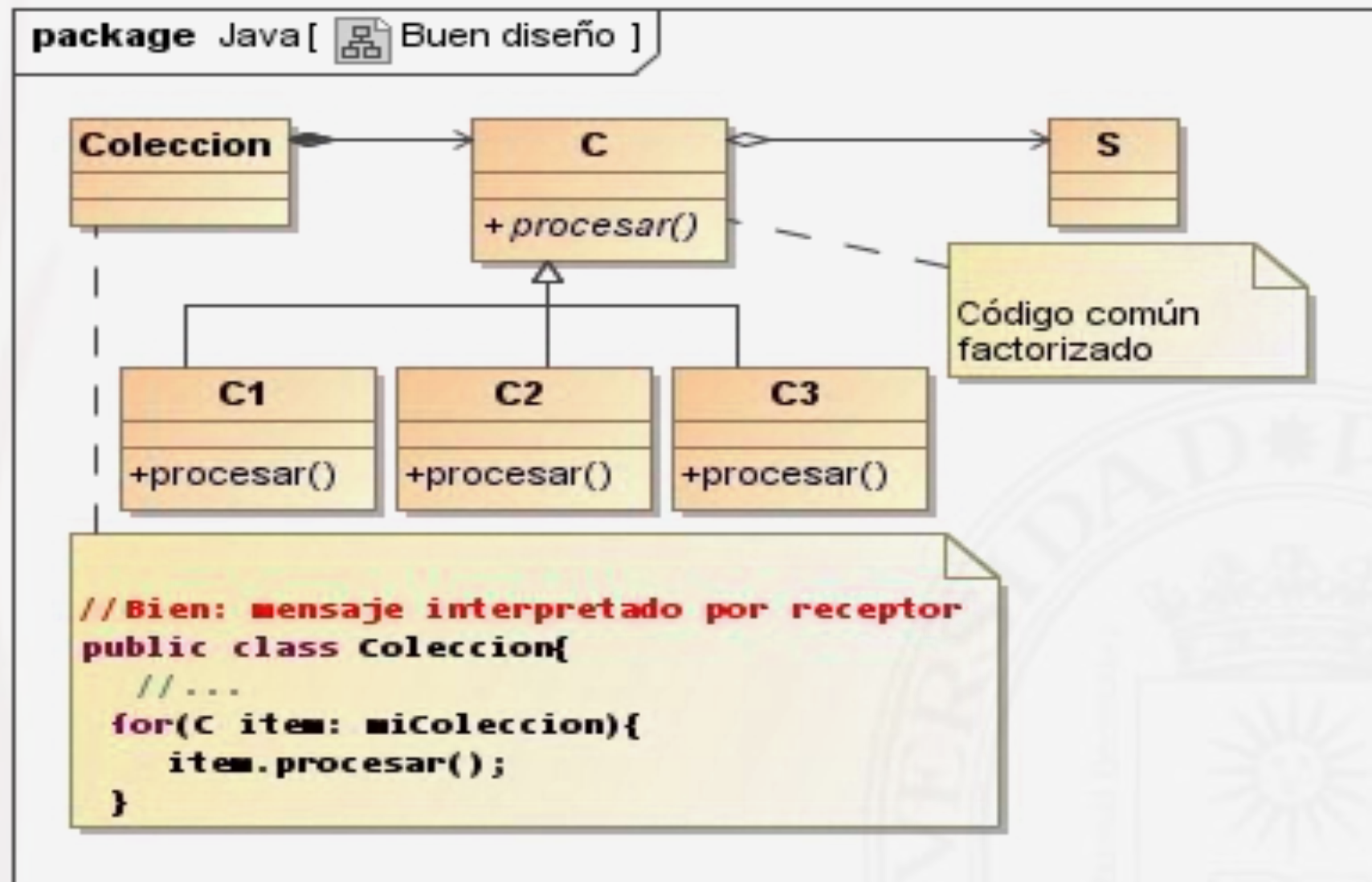
```
public class Agenda{  
    private Lista lista;  
  
    public Agenda() {  
        this.lista = new Lista();  
    }  
    //...  
}
```



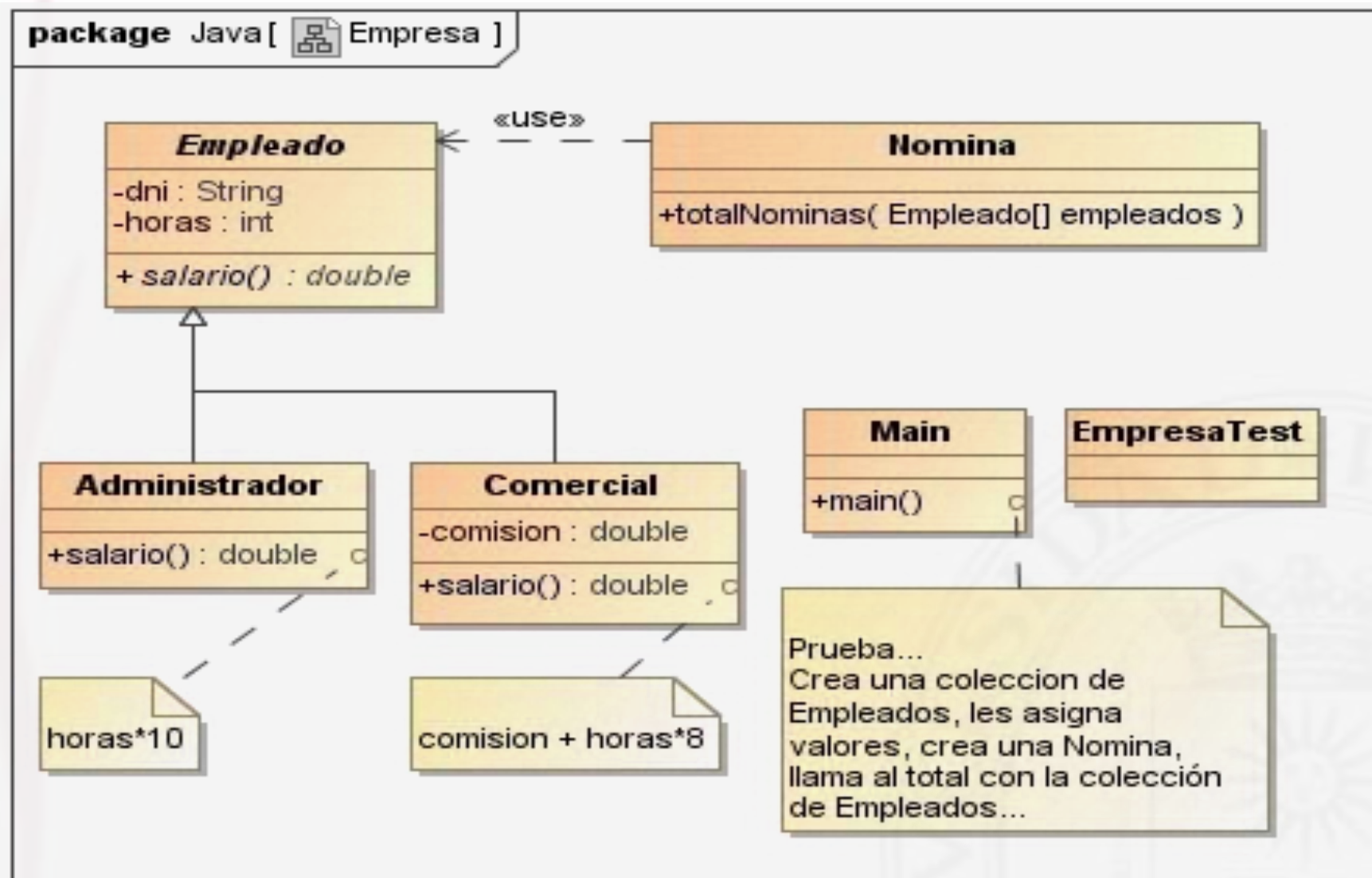
MAL Diseño



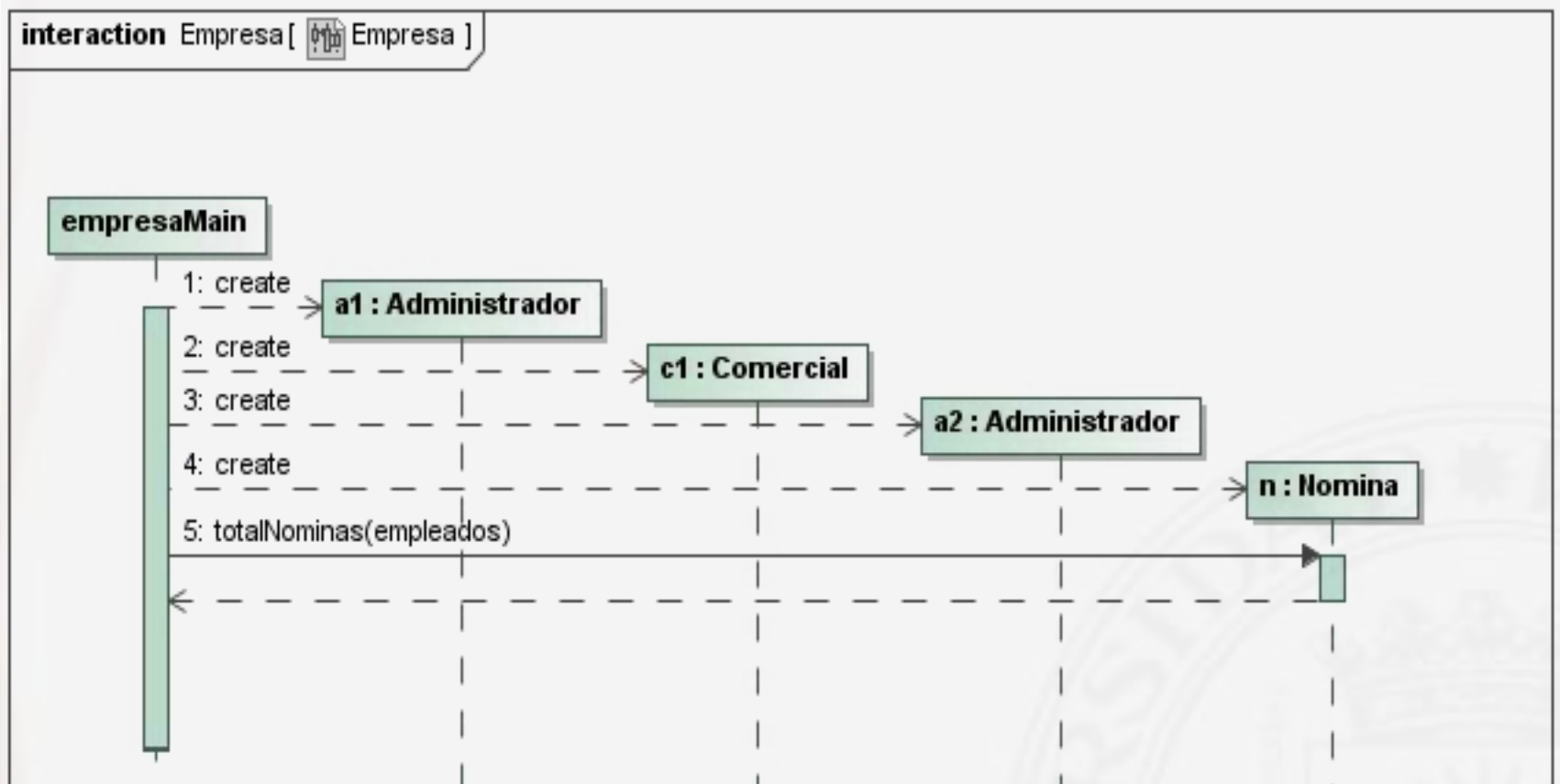
Buen Diseño



Aplicación Empresa



Aplicación Empresa



Aplicación Operandos

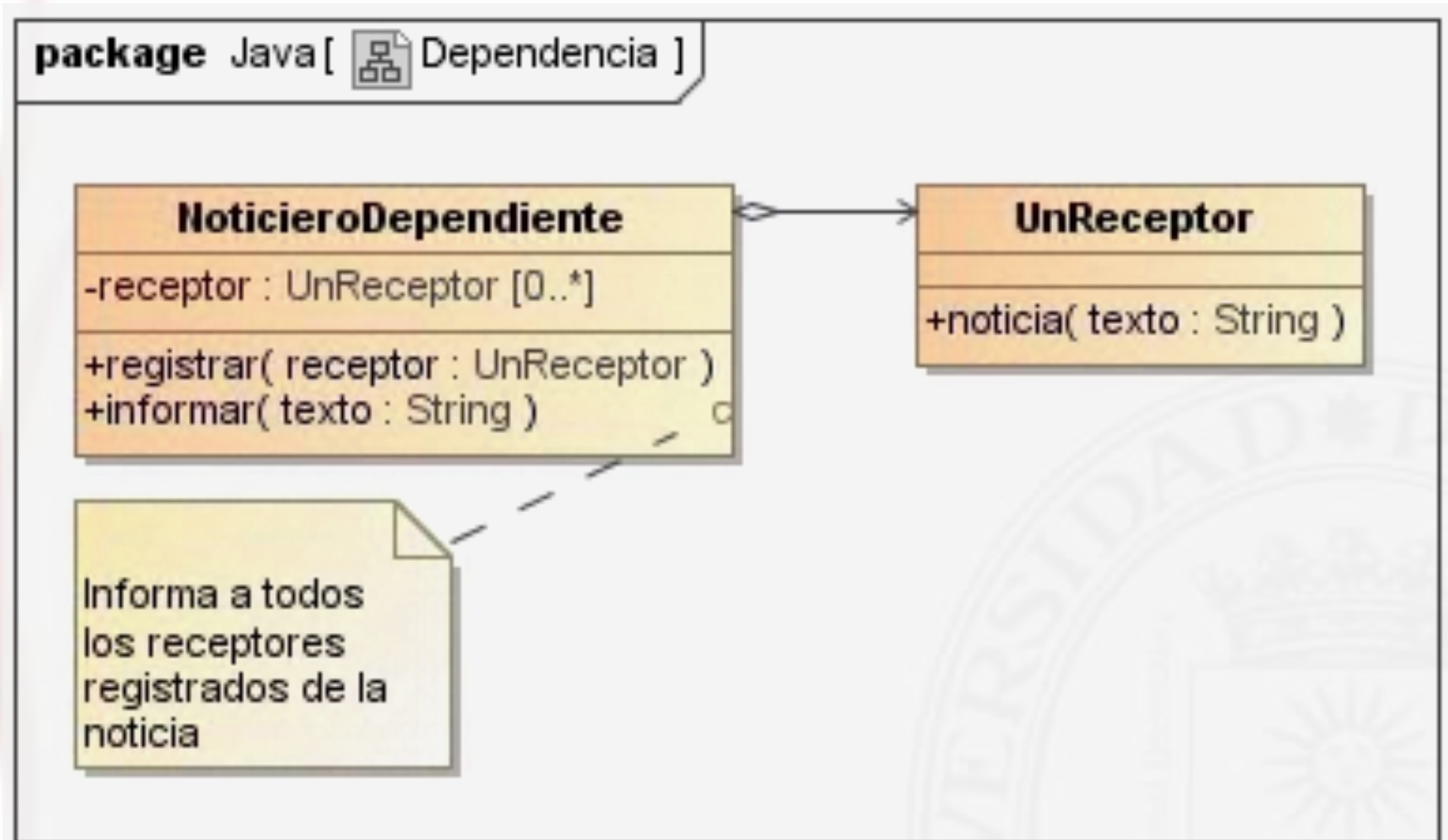
```
public class Suma {  
    private final int val1, val2;  
  
    public Suma(final int val1, final int val2) {  
        this.val1 = val1;  
        this.val2 = val2;  
    }  
    public int getVal1() { return this.val1; }  
    public int getVal2() { return this.val2; }  
    public int sumar() {  
        return this.val1 + this.val2;  
    }  
}
```

```
public class Resta {  
    private final int val1, val2;  
  
    public Resta(final int val1, final int val2) {  
        this.val1 = val1;  
        this.val2 = val2;  
    }  
    public int getVal1() { return this.val1; }  
    public int getVal2() { return this.val2; }  
    public int restar() {  
        return this.val1 - this.val2;  
    }  
}
```

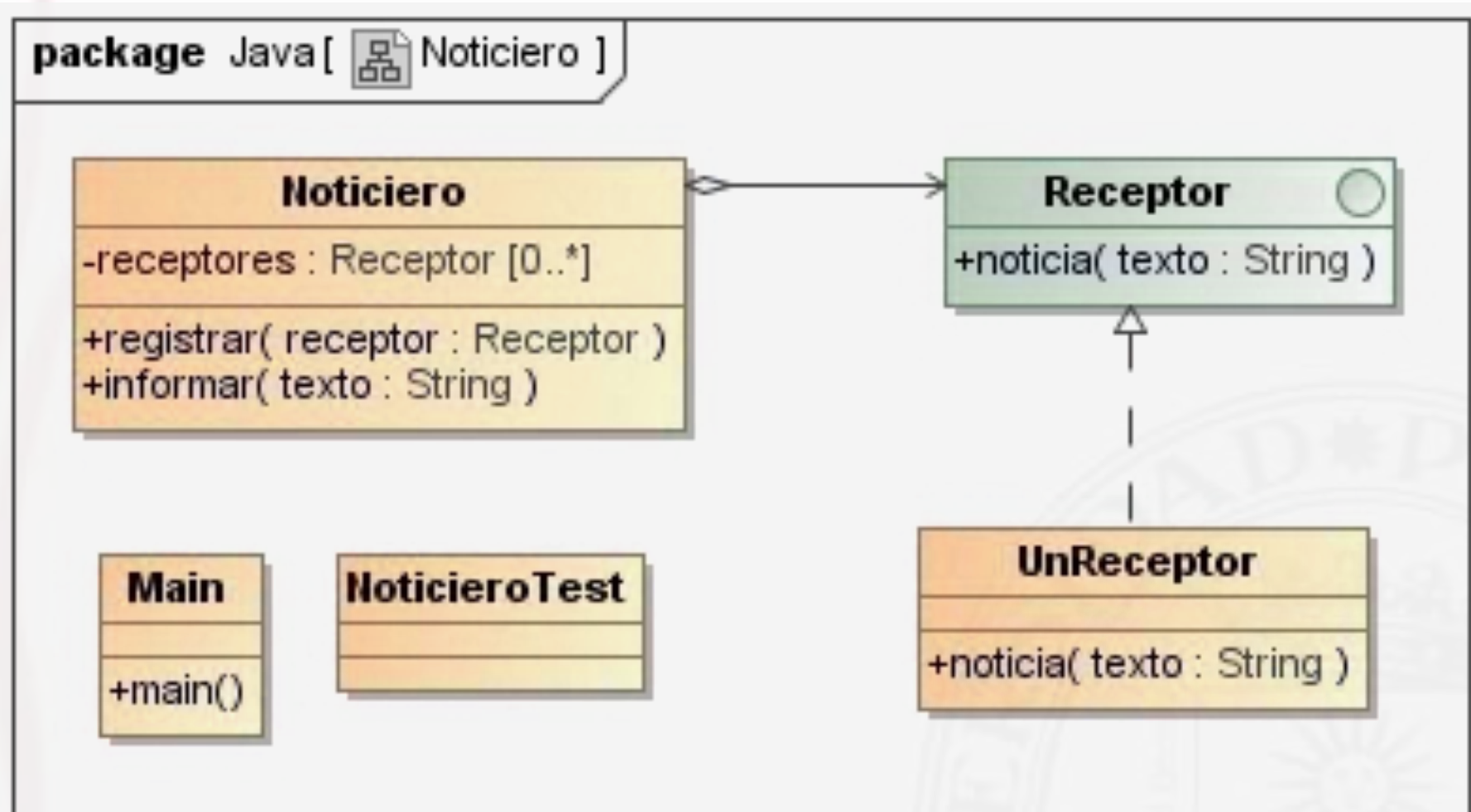
Aplicación Operandos

```
public class Operando {  
  
    // MAL DISEÑADO... MAL CODIFICADO  
    public int total(Object[] operandos) {  
        int result = 0;  
        for (Object operando : operandos) {  
            if (operando.getClass().getName().equals("Suma")) {  
                result += ((Suma) operando).sumar();  
            } else if (operando.getClass().getName().equals("Resta")) {  
                result += ((Resta) operando).restar();  
            }  
        }  
        return result;  
    }  
}
```

Dependencias



Dependencias



Paquetes

- Los paquetes agrupan un conjunto de clases que trabajan conjuntamente sobre el mismo ámbito
- Nombres: empieza por minúsculas y separadas por puntos
- Sentencia: `package` (antes de la clase)
- Estructura de carpetas de clases
- `paq.paq.Clase` o sentencia `import` (antes de la clase)
- `java.lang.String...` `java.lang` se importa automáticamente

Paquetes. Ejemplos

```
package p1.p2;  
public class MiClase{}
```

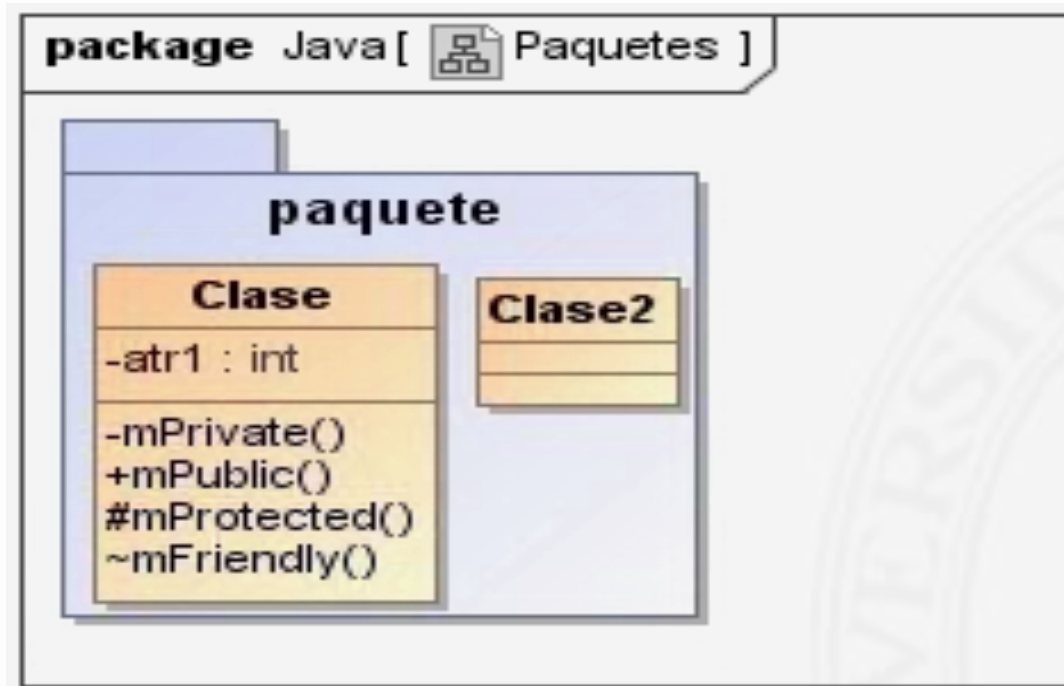
```
package p1;  
public class Clase2{}
```

```
public class Una{  
    private java.util.Map mapa;  
    public Una(){  
        this.mapa = new java.util.HashMap();  
        this.mapa.put("clave", "valor");  
    }  
}
```

```
import java.util.Map;  
import java.util.*; //importa todas las clases del paquete, NO los subpaquetes.  
public class Una{  
    private Map mapa;  
    ...
```

Visibilidad

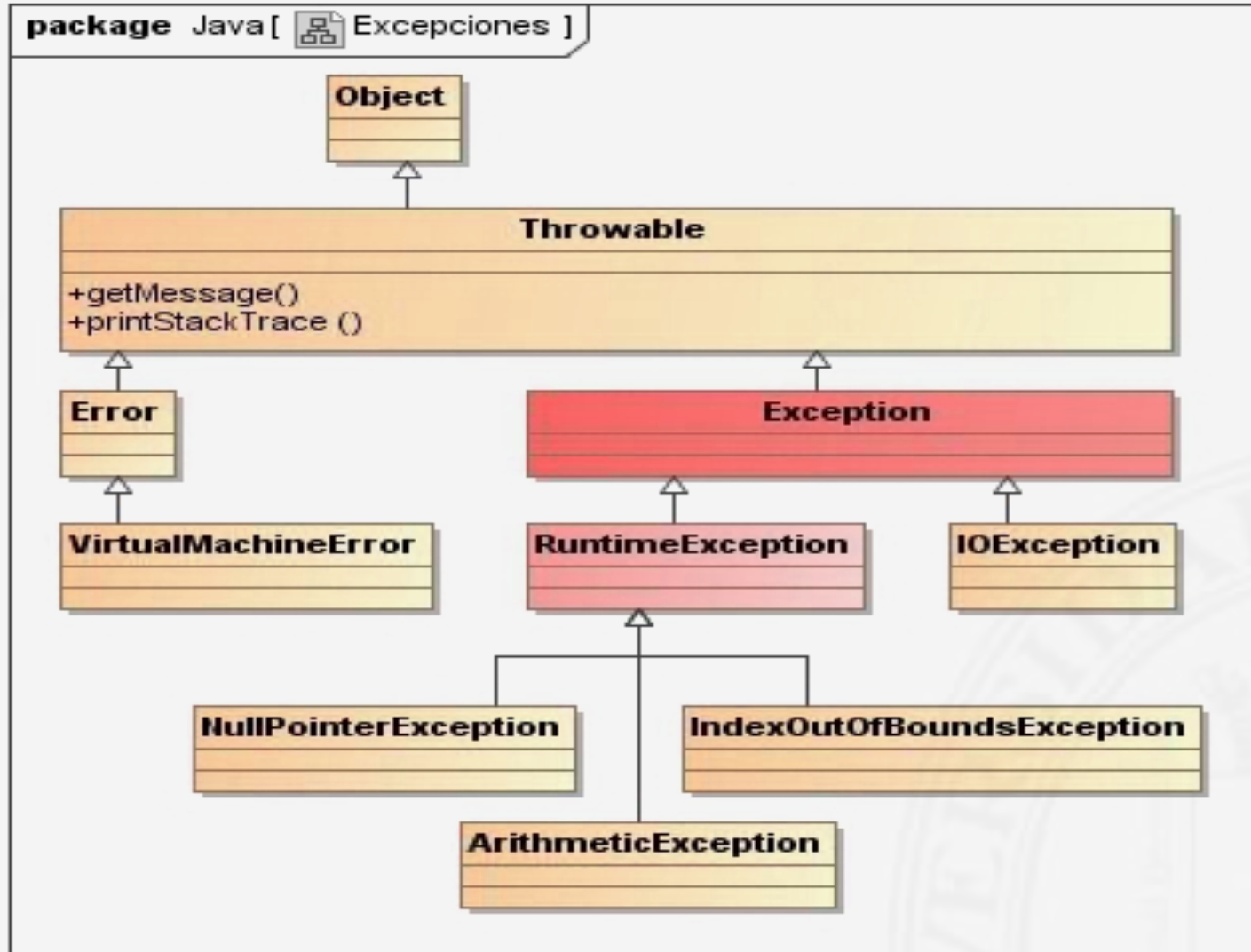
- **public:** visible para todos
- **private:** uso interno
- **protected:** es visible dentro del mismo paquete o cualquier subclase de forma directa (mediante super.elemento)
- **sin calificar (friendly):** es visible dentro del mismo paquete



Excepciones

- Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias
- Ya no hará falta preguntar a lo largo de nuestro programa si ha existido algún error; cuando se produce una excepción, se lanza automáticamente el código que la procesa
- Cuando un método lanza una excepción, se crea un objeto de la clase o subclases de *Exception*, que contiene los detalles de la excepción
- El sistema busca hacia atrás en la pila de llamadas, empezando por el método en el que ocurrió el error, hasta que encuentra un método que contiene el *manejador de excepción* adecuado
- Muchas clases de errores pueden utilizar excepciones, desde problemas de hardware, como la avería de un disco duro, a los simples errores de programación, como tratar de acceder a un elemento de un array fuera de sus límites
- Podemos agrupar las excepciones en tres tipos fundamentales:
 - **Error.** Excepciones que indican problemas muy graves asociados al funcionamiento de la máquina virtual y suelen ser no recuperables. Por ejemplo, se ha quedado sin memoria del sistema
 - **Exception.** Indican que un mensaje lanzado a un objeto no ha podido realizar su acción. Es una filosofía de programación. Por ejemplo, fichero que no existe. Deben ser capturadas
 - **RuntimeException.** Es una Excepción asociada a una mala programación, por ejemplo que un índice está fuera del rango del array. Estas excepciones no deben ser capturadas, sino que se debe corregir el problema de programación

Excepciones



Manejo de excepciones

- Cuando el programador va a ejecutar una porción de código que pueda provocar una excepción (*Exception*), puede actuar de dos maneras:
 - Tratando la excepción. Se debe incluir el código dentro de un bloque *try-catch*
 - Pasársela al método invocador. Se añade la palabra reservada *throws* a la definición del método.
- Hay que dejar claro que las excepciones (*RuntimeException*) causadas por una mala programación (por ejemplo, un índice fuera de límites) se deben corregir y no se deben tratar. Las excepciones que se tratan son aquellas que se lanzan por ser parte del funcionamiento previsto y controlado de un objeto, por ejemplo, error en la lectura de un fichero

Manejo de excepciones

```
try {  
    new FileReader("fichero");  
    // ...  
} catch (FileNotFoundException fnfe) {  
    System.out.println(fnfe.getMessage());  
} catch (Exception e) { // igual o más general que la anterior  
    System.out.println(e.getMessage());  
} finally {  
    System.out.println("Siempre se ejecuta");  
}
```

```
public void mException() throws FileNotFoundException {  
    new FileReader("fichero");  
    // ...  
}
```


Lanzamiento de excepciones

- El programador podrá lanzar excepciones de forma explícita. Nos podemos plantear dos causas para provocar excepciones:
 - Se lanza un mensaje a nuestro objeto que intenta realizar una acción no permitida. En este caso, se lanza una excepción de la clase o subclase de *RuntimeException*. Por ejemplo, en la clase *Natural* se intenta establecer un valor negativo
 - Se lanza un mensaje a nuestro objeto que no podemos garantizar que llegue a buen fin. Se debe definir la sentencia *throws* en la definición del método. Por ejemplo, se intenta mandar datos un servidor en Internet que podría no responder, en este caso, queremos asegurarnos que el método que invoca sea consciente que la acción puede fallar, y por lo tanto le obligamos a realizar un *try-catch*
- Se sobrecarga el sistema y se complica el lanzamiento de mensajes, conviene no abusar de esta técnica. Otra forma, con menos carga, sería devolver un *boolean* indicando si ha ido bien

```
public void metodo1() throws Exception {  
    if (condicion) throw new Exception("Descripción");  
}  
public void metodo2() {  
    if (condicion) throw new RuntimeException("Descripción");  
    if (condicion) throw new NumberFormatException("Descripción");  
}
```


Aserciones

- Se permite comprobar una suposición, si no se cumple se lanza una excepción. Sólo se activan en la fase de desarrollo. En la fase de explotación se supone que se cumple y no se comprueba
 - `assert expBooleana;`
 - `assert expBooleana:exp;`
- Para ejecutar con aserciones:
 - `java -ea Miclase`
 - Eclipse: Run Configurations... → Arguments → VM arguments → `-ea`

```
public void mtd(final String str) {  
    assert str != null : "str no puede ser null";  
    if ("uno".equals(str)) {  
        System.out.println("uno");  
    } else if ("dos".equals(str)) {  
        System.out.println("dos");  
    } else {  
        assert false : "No deberíamos estar aquí";  
    }  
}
```

Aserciones. Cuándo utilizarlos

- No utilizar en el control de parámetros de métodos públicos. Futuros usos no controlados
 - No en librerías públicas
 - Si en clases del proyecto propio
- No utilizar en validar datos de entrada (GUI)
- Si utilizar en else-if o switch de casos imposibles (último else)
- Si utilizar en código inalcanzable: `assert false: “no deberíamos estar aquí”;`
- Si utilizar en comprobar la lógica del código

Tipos Genéricos

- Los tipos genéricos, también llamados tipos parametrizados, permiten definir un tipo mediante un parámetro, que se le dará valor concreto en la instanciación
- Los tipos parametrizados se utilizan especialmente para implementar tipos abstractos de datos: pilas, colas... y otros que permiten almacenar distintos tipos de elementos
- Sin los tipos genéricos, nos vemos a utilizar el tipo *Object*. Teniendo el inconveniente de realizar *casting* para recuperar los datos y perdiendo la capacidad el compilador de realizar un control de tipado
- Para agregar a una clase tipos genéricos, se le añade después del nombre una lista de parámetros que representan tipos. El formato es el siguiente:
 - **public class** MiClase<tipo1, tipo2...> {}
- Aunque podemos utilizar cualquier nombre de parámetros para contener tipos, se recomienda los siguientes nombres descriptivos:
 - E. Elemento
 - K. Clave
 - N. Número
 - T. Tipo
 - V. Valor
 - S,U,V... para 2º, 3º... tipos

Tipos Genéricos

```
public class MiGenerico<K, V> {  
    private K atributo1;  
    private V atributo2;  
  
    public MiGenerico(K atributo1, V atributo2) {  
        this.atributo1 = atributo1;  
        this.atributo2 = atributo2;  
    }  
    public K getAtributo1() {  
        return atributo1;  
    }  
    public V metodo(K parametro1) {  
        return this.atributo2;  
    }  
}
```

```
MiGenerico<Integer, String> mc = new MiGenerico<Integer, String>(0, "");  
String res = mc.metodo(3); // Se aplica boxing para convertir 3 en: new Integer(3)
```

Enumerados

- Los tipos enumerados son un tipo definido por el usuario, que representa un conjunto de constantes. Al ser constantes los valores, se deben escribir en mayúsculas
- Los enumerados se pueden definir en:
 - Un fichero independiente (son una clase especial)
 - Dentro de una clase, en la zona de atributos, pero no se pueden definir dentro de los métodos
- Para utilizarlo, se referencia como si fueran constantes estáticas de clase
- Existen acciones que se pueden realizar dentro de un *enum*. Nos ofrecen dos funciones importantes:
 - *valueOf(String s)*, este método te devuelve una referencia de la constante del enumerado que coincide con el String pasado por parámetro
 - *values()*, este método te devuelve un array con todos las constantes del enumerado

Enumerados

```
public enum Semana {  
    LUN, MAR, MIE, JUE, VIE, SAB, DOM  
}
```

```
public class Clase{  
    public enum Semana { LUN, MAR, MIE, JUE, VIE, SAB, DOM};  
    //...  
}
```

```
Semana dia;  
dia = Semana.MAR;  
dia = Semana.valueOf("MAR");
```

Enumerados

```
public enum Planeta {  
    MERCURIO(3.303e+23, 2.4397e6),  
    VENUS(4.869e+24, 6.0518e6),  
    TIERRA(5.976e+24, 6.37814e6),  
    MARTE(6.421e+23, 3.3972e6),  
    JUPITER(1.9e+27, 7.1492e7),  
    SATURNO(5.688e+26, 6.0268e7),  
    URANO(8.686e+25, 2.5559e7),  
    NEPTUNO(1.024e+26, 2.4746e7),  
    PLUTON(1.27e+22, 1.137e6);  
    // constante universal de gravedad  
    public static final double G = 6.67300E-11;  
    public final double masa; // in kilogramos  
    public final double radio; // in metros  
  
    private Planeta(double mass, double radius) {  
        this.masa = mass;  
        this.radio = radius;  
    }  
    public double gravedadSuperficie() {  
        return G * this.masa / (this.radio * this.radio);  
    }  
    public double pesoSuperficie(double masa) {  
        return masa * this.gravedadSuperficie();  
    }  
}
```

Colecciones. Collection

- *Es el interface más importante*
 - *add(T e)*. Añade un elemento
 - *clear()*. Borra la colección
 - *remove(Object obj)*. Borra un elemento
 - *isEmpty()*. Indica si está vacía
 - *iterator()*. Devuelve un *Iterator* de la colección, útil para realizar un recorrido de uno en uno.
 - *size()*. Devuelve el tamaño de la colección
 - *contains(Object e)*. Nos indica si el objeto está contenido. Atención, si el objeto tiene sobrescrito el método *equals* o *hashCode* se utilizarán, sino se utilizarán las versiones de la clase *Object*.
 - *toArray(...)*. Devuelve una array con el contenido de la colección
- Para recorrerla: *for each*

Colecciones. Iterator

- Es un objeto que te permite recorrer la colección
 - *hasNext()*. Devuelve true si existen más elementos
 - *next()*. Devuelve el siguiente elemento
 - *remove()*. Borrar el elemento de la colección

```
public void iterador() {  
    final Collection<String> lista = new ArrayList<String>();  
    lista.add("uno");  
    lista.add("");  
    lista.add("dos");  
    for (String item : lista) {  
        System.out.println(item);  
    }  
    final Iterator<String> itr = lista.iterator();  
    while (itr.hasNext()) {  
        final String str = itr.next();  
        if ("".equals(str)) {  
            itr.remove();  
        }  
    }  
}
```

Colecciones. List

- Es una lista, no ordenada, en que se mantiene el orden de los elementos, pudiendo acceder a su contenido según la posición. Esta lista crece según las necesidades. Este interface hereda de *Collection*, y añade los siguientes métodos:
 - `add(int index, E element)`. Se añade en una posición determinada
 - `get(int index)`. Se recupera el elemento de una posición
 - `set(int index, E element)`. Reemplaza el elemento de una posición
- *Object: equals*. Utiliza este método para comparar
- Podemos destacar las siguientes implementaciones:
 - *ArrayList*. Esta implementación mantiene la lista compactada en un array. Tiene la ventaja de realizar lecturas muy rápidas, el problema está en borrar elementos intermedios. Esta clase no está preparada para trabajar con varios hilos; para ello tenemos la implementación *Vector*
 - *Vector*. Es muy parecida al anterior, pero con la diferencia de estar preparada para trabajar con hilos
 - *LinkedList*. Esta lista se implementa mediante una lista entrelazada, formada por nodos que apuntan al elemento siguiente y el elemento anterior. Esta implementación favorece las inserciones y el borrado, pero hacen muy lento el recorrido

Colecciones. Set

- Es una colección en la que no contiene elementos duplicados. Hereda del interface `Collection`, apenas añade métodos nuevos.
- Podemos destacar la implementación *HashSet*:
 - Esta implementación se basa en una tabla hash. Nos referimos a hash, como una función que genera claves, garantizando que el mismo elemento genera siempre la misma clave. Para obtener la clave, se aplica la función hash al valor devuelto por el método *hashCode* de `Object`. Utilizando esta clave como índice de una array se puede obtener accesos muy rápidos a los objetos. Es posible que dos objetos diferentes generen la misma clave
 - No se garantiza que se mantenga el orden a lo largo del tiempo. Pero ofrece un tiempo constante en la ejecución de las operaciones básicas: `add`, `remove`, `contains` y `size`
 - Se debe tener cuidado porque en esta implementación se consideran que dos objetos son diferentes si su *hashCode* es diferente, aunque el método `equals` devuelva `true`. Por ello, puede que debamos sobrescribir el método *hashCode* de `Object`.

Colecciones. SortedSet

- Esta colección mantiene un orden entre los elementos. Para poder mantener el orden, los objetos deben implementar el interface Comparable, el cual tiene un único método (*compareTo*). Con este método implementamos un orden entre nuestros objetos
- Podemos destacar la implementación *TreeSet*
 - Mantiene los objetos ordenados en un árbol binario balanceado. Realiza búsquedas relativamente rápidas, aunque más lentas que un *HashSet*. Esta implementación no está preparada para trabajar con hilos

Colecciones. Queue

- Esta colección está pensada para organizar una cola (FIFO). Los elementos se añaden por el final, y se extraen por el principio. Dispone de los siguientes métodos:
 - `boolean add(E e)`. Inserta un elemento en la cola, provoca una excepción si no existe espacio disponible.
 - `E element()`. Recupera sin borrar la cabeza de la cola, si está vacía provoca una excepción
 - `boolean offer(E e)`. Inserta, si puede, un elemento en la cola.
 - `E peek()`. Recupera sin borrar la cabeza de la cola, si está vacía devuelve null
 - `E poll()`. Recupera y borra la cabeza de la cola, o devuelve null si está vacía
 - `E remove()`. Recupera y borra la cabeza de la cola, si está vacía provoca una excepción
- La clase *LinkedList* también implementa esta interface.
- Existe el interface *Deque*, que implementa una cola doble, útil para realizar *LIFO*. *LinkedList* y *ArrayDeque* implementan este interface.

Colecciones. Map

- Un *Map* es una colección de duplas de clave-valor, también conocido como diccionario. Las claves no pueden estar repetidas, si dos elementos tienen la misma clave se considera que es el mismo. *Map* no hereda de *Collection*.
- Algunos de sus métodos son:
 - `boolean containsKey(Object key)`. Devuelve `true` si contiene la clave
 - `boolean containsValue(Object value)`. Devuelve `true` si tiene este valor
 - `V get(Object key)`. Devuelve el valor asociado a la clave
 - `Set<K> keySet()`. Devuelve un *Set* con las claves del *Map*
 - `V put(K key, V value)`. Asocia un valor a una clave
 - `V remove(Object key)`. Elimina un valor
 - `Collection<V> values()`. Devuelve una colección con todos los valores
- Tiene varias implementaciones, una interesante es *HashMap*.

Colecciones. SortedMap

- Es un *Map* ordenado basado en el interface *Comparable*. Esta ordenado según las claves. Si se utilizan las clases *Integer*, *Double*, *String*... como claves, ya implementan el interface *Comparable*.
- Tiene varias implementaciones, una interesante es *TreeMap*.

Ejercicios. Gestión de vehículos para alquilar

- Implementar el modelo de clases para gestionar un conjunto de vehículos, existen tres tipos:
 - Coches:
 - Identificador, único para cada vehículo, independiente del tipo
 - Descripción, una cadena que describe al vehículo
 - Categoría, existen 3 categorías (A, B, C)
 - Precio de alquiler depende de la categoría (10, 15 y 20€) y de los días alquilados: de 1º al 3º día es el precio base, del 4º al 7º día es 80% del precio base, y el resto de días al 50% del precio base
 - Motos:
 - Identificador, único para cada vehículo, independiente del tipo
 - Descripción, una cadena que describe al vehículo
 - Precio de alquiler es de 8€ / día si se alquila menos de una semana y de 7€/ día si se alquila más de una semana
 - Bicicletas:
 - Identificador, único para cada vehículo, independiente del tipo
 - Descripción, una cadena que describe al vehículo
 - Precio es de 3€/día los primeros 2 días, y de 2€ día el resto
- Las funcionalidades que se ofrecen son:
 - Dar de alta un nuevo vehículo
 - Ver un String con todos los vehículos
 - Dado un identificador de vehículo y unos días de alquiler, dar el precio

Ejercicios. Gestión de vehículos para alquilar

```
public class IOMain {
    // Atributos...

    public void accion1() {
        Clase c = (Clase) IO.in.read("paquete.Clase", "Clase");
        // ...
    }

    public void accion2() {
        Clase2 c2 = (Clase2) IO.in.read("paquete.Clase2", "Clase2");
        // ...
    }

    public void ver() {
        IO.out.println("accion...");
    }

    public void precio() {
        int id = IO.in.readInt("id");
        int dias = IO.in.readInt("dias");
        IO.out.println("accion...");
    }

    public static void main(String[] args) {
        IO.in.addController(new IOMain());
    }
}
```

Ejercicio. Biblioteca

- Se pide desarrollar una aplicación que permita gestionar una biblioteca. Cada libro se registra con su ISBN, título y autor. Sólo existe un ejemplar por cada libro. Un libro puede estar prestado, con el usuario del préstamo, fecha de inicio y fecha de fin.
- Las funcionalidades mínimas son:
 - Dar de alta/baja un libro
 - Dar de alta/baja un usuario
 - Mostrar todos los libros
 - Mostrar todos los usuarios
 - Mostrar datos del libro
 - Conocer si un libro está libre o prestado
 - Prestar/devolver un libro

Ejercicio: Gestión de reservas de un Hotel

- Se trata de un sistema de reservas de un hotel. El hotel permite hacer reservas de habitaciones y de ello se encarga el Administrativo de Reservas, y de forma alternativa el cliente la puede hacer también por Internet. Siempre que se solicita la reserva de una habitación, se comprueba si la persona es cliente del hotel, en cuyo caso solo hay que cumplimentar los datos de la reserva (fecha de entrada, fecha de salida, tipo de habitación, etc.). Si la persona no es cliente del hotel, además, hay que darlo de alta e introducir sus datos personales. Una reserva puede ser anulada en cualquier momento, pero si se realiza con menos de 24 horas de antelación, se penaliza al cliente cargándole una cantidad equivalente al 50% del coste de una noche en la habitación reservada. Una vez el cliente llega al hotel, el Recepcionista comprueba su reserva y los datos del cliente y procede a registrarlo. Una vez concluida la estancia, el Recepcionista procede a facturar la estancia.