



<http://vios.dc.fi.udc.es/tp/>

Introducción a la programación con JAVA

Elena Hernández Pereira
Óscar Fontenla Romero
Antonio Fariña

...

Departamento de Computación
Facultad de Informática
Universidad de A Coruña

Tecnología de la Programación
– Febrero 2008 –

Sumario

- Programación orientada a objetos (POO)
 - Encapsulación (de operaciones y estado: visibilidad de métodos y/o atributos)
 - Herencia
 - Polimorfismo

- Lenguaje de programación Java



Programación orientada a objetos (POO)

Programación orientada a objetos: POO (I)

- La programación orientada a objetos se basa en la programación de clases
- Un programa se construye a partir de un conjunto de clases
- **Clase**: una agrupación de **datos** (variables) y de **funciones** (métodos) que operan sobre los datos.
- A estos datos y funciones pertenecientes a una clase se les denomina **variables** y **métodos** o **funciones miembro**
- Todos los métodos y variables se definen dentro del bloque de la clase

Programación orientada a objetos: POO (II)

- Conceptos importantes de la POO:
 - **Encapsulación:**
 - Las estructuras de datos y los detalles de la implementación de una clase se hallan **ocultos** de otras clases del sistema
 - **Control de acceso** a variables y métodos
 - Acceso público / privado / ...

Programación orientada a objetos: POO (III)

- Conceptos importantes de la POO:
 - **Herencia:**
 - Una clase (subclase) puede derivar de otra (superclase)
 - La subclase hereda todos los atributos y métodos de la superclase
 - Las subclase puede redefinir y/o añadir atributos y métodos
 - Fomenta la reutilización de código

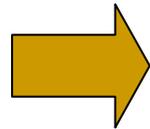
Programación orientada a objetos: POO (IV)

- Ejemplo herencia:

Clase Trabajador



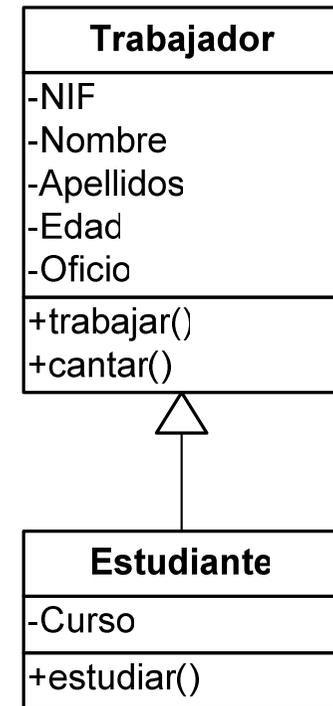
NIF
Nombre
Apellidos
Edad
Oficio



Clase ALUMNO (SUBCLASE)



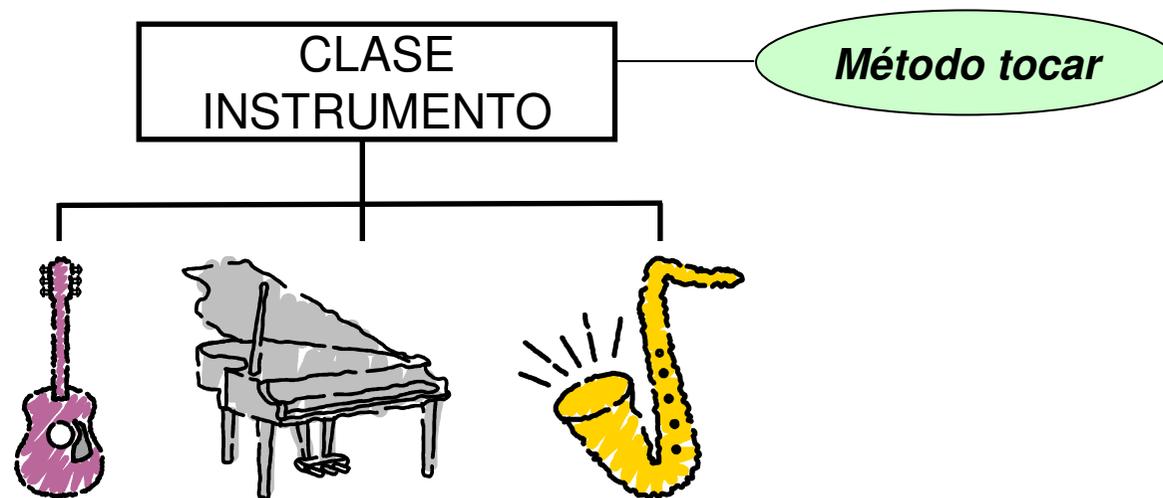
Curso



Programación orientada a objetos: POO (V)

□ Polimorfismo:

- Es la capacidad de tener métodos con el mismo nombre (y argumentos) y *diferente implementación*
- Una operación puede tener más de un método que la implementa



Clases y objetos

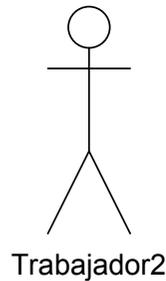
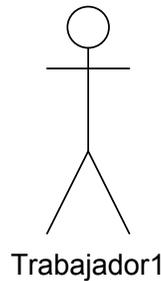
- Una vez definida e implementada una clase, es posible declarar elementos de esta clase: **objetos**
- De una única clase se pueden declarar o crear numerosos **objetos**.
- La **clase** es lo genérico: es el patrón o modelo para crear **objetos**.
- Cada objeto tiene sus propias copias de las variables miembro, con sus propios valores
- Vista externa de una clase: **interfaz**
 - Atributos y métodos visibles por otras clases

Clases y objetos

- Ejemplo

```
Trabajador trabajador1 = new Trabajador();  
trabajador1.nombre = "Pepe";  
Trabajador trabajador2 = new Trabajador();  
trabajador2.nombre = "María";
```

Trabajador
-NIF
+Nombre
-Apellidos
-Edad
-Oficio
+trabajar()
+cantar()





Programación con Java

Bibliografía (I)

- B. Eckel. **Piensa en Java** (2ª edición). Prentice Hall. 2002
- H.M. Deitel, P.J. Deitel. **Cómo programar en Java** (5ª edición). Pearson Prentice-Hall. 2004
- D. Arnow, G. Weiss, C.-Brooklyn. **Introducción a la programación en Java. Un enfoque orientado a objetos.** Pearson Addison Wesley. 2000
- K. Arnold, J. Gosling, D. Holmes. **El lenguaje de programación JAVA** (3ª edición). Pearson Addison Wesley. 2001

Bibliografía (II)

- J. Jalón, J.I. Rodríguez, I. Mingo, A. Imaz, A. Brazález, A. Larzabal, J. Calleja, J. García. **Apreda java como si estuviese en primero**. Escuela Superior de Ingenieros Industriales. Universidad de Navarra <http://mat21.etsii.upm.es/ayudainf/>
- Fco. J. Ceballos, **Java 2. Curso de programación**, Ra-Ma, 2000
- Y mucha más documentación en Internet (tutoriales, cursos, ...)

Introducción (I)

- Desarrollado por Sun Microsystems en 1995
- Características:
 - **Simple**
 - Código similar a C/C++ pero eliminando algunos elementos conflictivos: punteros, herencia múltiple, etc.
 - **Portable**
 - Representación y comportamiento único para los tipos primitivos
 - Sistema abstracto de ventanas que presenta el mismo comportamiento en distintos entornos
 - **Multiplataforma**
 - JVM disponibles para "todos" los SO.

Introducción (II)

- Características (continuación):
 - **Robusto**
 - Fuerte comprobación de tipos y de límites de los arrays
 - Ausencia de punteros
 - Manejo de errores (excepciones)
 - **Seguro**
 - No se puede acceder a memoria directamente mediante punteros
 - Gestor de seguridad (Security Manager) para los bytecodes
 - **Orientado a objetos "puro"**
 - Obliga a trabajar en términos que facilitan la reutilización

Introducción (III)

- Características (continuación):
 - **Orientado a Internet**
 - Ej: servlets → jsp, applets,...
 - **Multihilo (multi-thread)**
 - Da soporte a la programación de procesos concurrentes
 - **Dinámico**
 - Permite la carga dinámica de clases
 - Búsqueda de nuevos objetos o clases en entornos distribuidos
 - **Lenguaje interpretado**
 - **JVM**

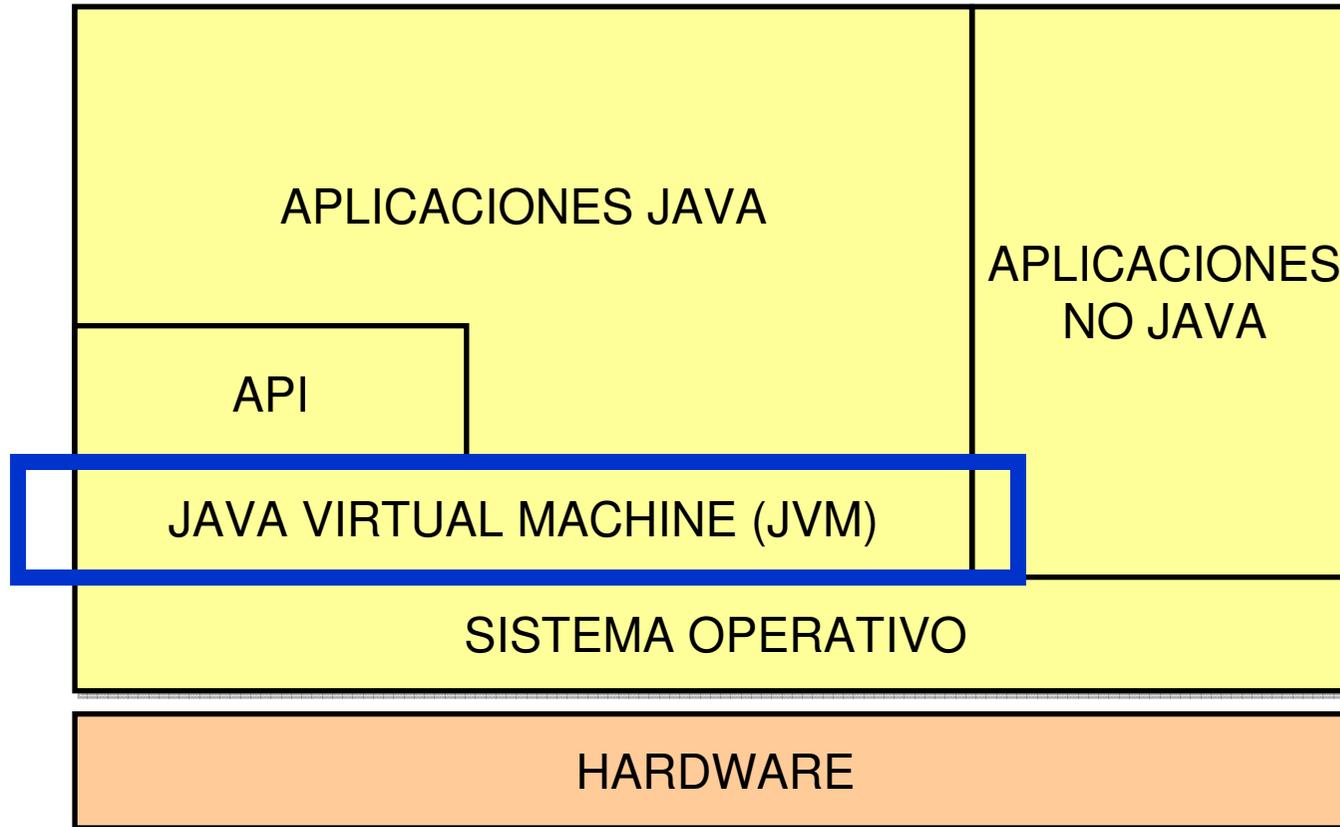
Introducción (IV)

- Inconvenientes:
 - Amplio número de plataformas a soportar
 - No todas soportan la misma versión del lenguaje
 - *Sun* sólo soporta las versiones de MS Windows y Solaris
 - ¿Lentitud? y amplio consumo de recursos
 - La máquina virtual está interpretando continuamente el ByteCode
 - Utilización de los elementos avanzados de la plataforma:
 - Recolector de basura, gestor de seguridad, carga dinámica de clases, comprobaciones en tiempo de ejecución, etc.
 - Ej: pero si yo sólo quería un programa "HolaMundo" !!!

Java Development Kit (JDK)

- Versiones:
 - 1995 - JDK 1.0
 - 1997 - JDK 1.1
 - 1998 - JDK 1.2 (Java 2)
 - 2000 - JDK 1.3
 - 2003 - JDK 1.4
 - 2004 - JDK 1.5
 - 2006 - JDK 1.6
- Java Runtime Environment (JRE)
 - Java Virtual Machine (JVM)
 - Java API: lenguaje básico + biblioteca estándar de clases

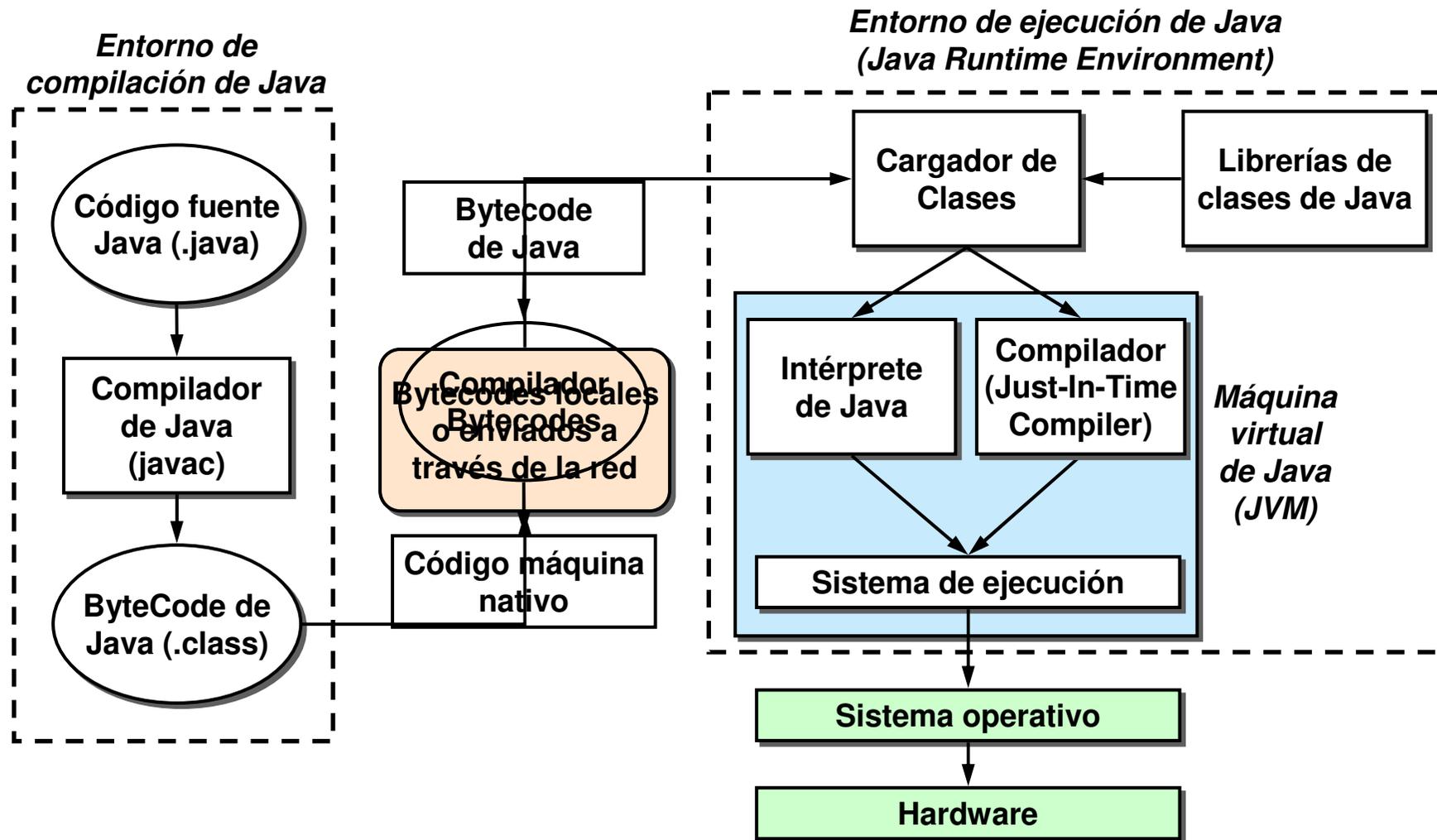
Modelo de ejecución



Compilación y ejecución (I)

- Compilador: **javac**
 - Código fuente → extensión `.java`
 - Ficheros compilados (bytecodes) → extensión `.class`
- Ejecución: **java**
 - Ejecuta los ficheros `.class`
- Herramienta de compresión: **jar**
 - Permite comprimir los ficheros compilados → extensión `.jar`
- Variable de entorno `CLASSPATH`: determina dónde se encuentran las clases de Java (del API y/o otros paquetes)

Compilación y ejecución (II)



Clases y objetos en Java (I)

- La clase consiste en:
 - **Atributos** (datos que contienen: variables)
 - **Métodos** (operaciones que se les puede aplicar)
- Un programa está compuesto por un conjunto de clases (al menos una)
 - Debe existir un método **main()** en una de ellas
- La clase define un determinado tipo de objetos
→ *abstracción*

Clases y objetos en Java (II)

- Definición:

```
class NombreClase  
{  
    // Atributos ...  
  
    // Métodos ...  
}
```

- Atributos → variables

- De tipo primitivo u otra clase

- Métodos → declaraciones de funciones:

- Contiene el código que se ejecutará cuando se invoque

Clases y objetos en Java (III)

- Ejemplo de una clase:

```
/** Clase de ejemplo: cuenta bancaria */  
class CuentaBancaria  
{  
    long numero;  
    string titular;  
    long saldo;  
  
    void ingresar (long cantidad)  
    {  
        saldo = saldo + cantidad;  
    }  
    void retirar (long cantidad)  
    {  
        if (cantidad <= saldo)  
            saldo = saldo - cantidad;  
    }  
}
```

Atributos

Métodos

Clases y objetos en Java (IV)

- Ejemplo de una clase:

Punto
-_x : double
-_y : double
+x() : double
+y() : double
+x (x:double)
+y (y:double)

```
class Punto /* Clase de ejemplo: Punto */
{
    double _x;           // atributo x
    double _y;           // atributo y

    double x() {         // método x()
        return _x;
    }
    double y() {         // método y()
        return _y;
    }
    void x(double d) {   // método x(double)
        _x = d;
    }
    void y(double d) {   // método y(double)
        _y = d;
    }
}
```

Clases y objetos en Java (V)

- Las clases anteriores no pueden ejecutarse por sí mismas
- Son sólo definiciones que permiten crear y manipular objetos de esa clase
- Arranque de un programa en java, en un clase especial del programa:
 - Contiene el método **main()** → comienza la ejecución del programa
- En un fichero fuente puede haber varias clases pero sólo una que contenga el método main()

Clases y objetos en Java (VI)

- **Declaración** de los objetos de una clase:
 - *Sintaxis:: NombreClase nombreObjeto;*
 - Ejemplo: Punto miPunto;
- **Creación** de los objetos de la clase:
 - Operador **new**
 - *nombreObjeto = new NombreClase();*
 - Ejemplo: miPunto = new Punto();
- Se pueden declarar y crear al mismo tiempo:
 - Punto miPunto = new Punto();

Punto
-_x : double
-_y : double
+x() : double
+y() : double
+x (x:double)
+y (y:double)

Clases y objetos en Java (VII)

- ¿Qué podemos hacer con el objeto?
 - Acceder a sus atributos y métodos
 - Para acceder se usa la **notación punto**:
 - nombreObjeto.característica;
 - Ejemplos:
 - miPunto._x
 - miPunto._y
 - miPunto.x() → deben incluirse los paréntesis
 - La ejecución de un método: *paso de mensaje*

Clases y objetos en Java (VIII)

- Ejemplo: fichero Prueba.java

Prueba.java

```
class Punto {
    double _x;

    double _y;

    double x(){
        return _x;
    }
    double y(){
        return _y;
    }
    void x(double d){
        _x = d;
    }
    void y(double d){
        _y = d;
    }
}
```

```
class Punto /* Clase Punto anterior */
{
    ...
}

public class Prueba
{
    // Programa principal
    public static void main (String [] args)
    {
        Punto p = new Punto();
        p.x(3.0);
        System.out.println("Coordenada x =" + p.x());
        //escribe 3 por pantalla.
    }
}
```

Clases y objetos en Java (IX)

- Tipos de variables:
 - Primitivas (entero, flotante, carácter, etc.)
 - De objeto (**Punto** p)
- Asignación en variables primitivas:
 - Realiza una copia de los valores (como siempre)

```
int numero1 = 12, numero2 = 18;  
numero2 = numero 1;
```

numero1	numero2
12	18

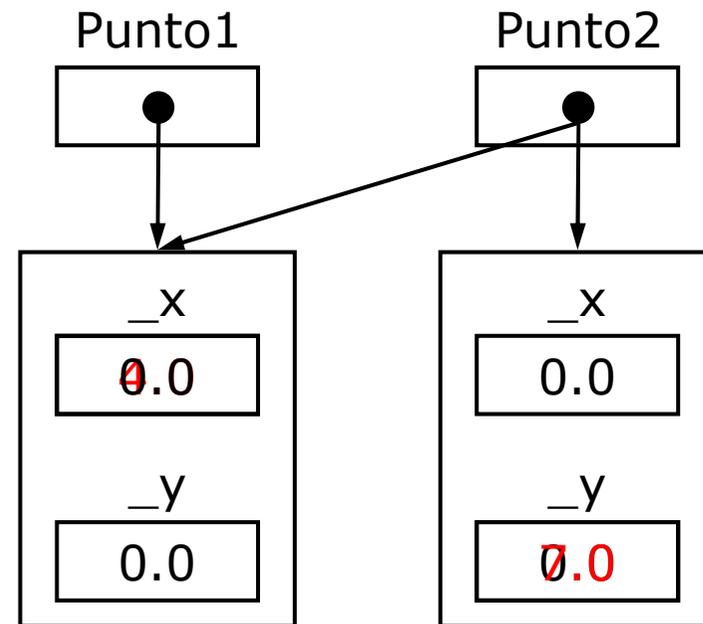
Clases y objetos en Java (X)

- Asignación en variables de objetos:
 - Son "*referencias*"

```
Punto punto1 = new Punto();  
Punto punto2 = new Punto();
```

```
Punto1.x(4);  
Punto2.y(7);
```

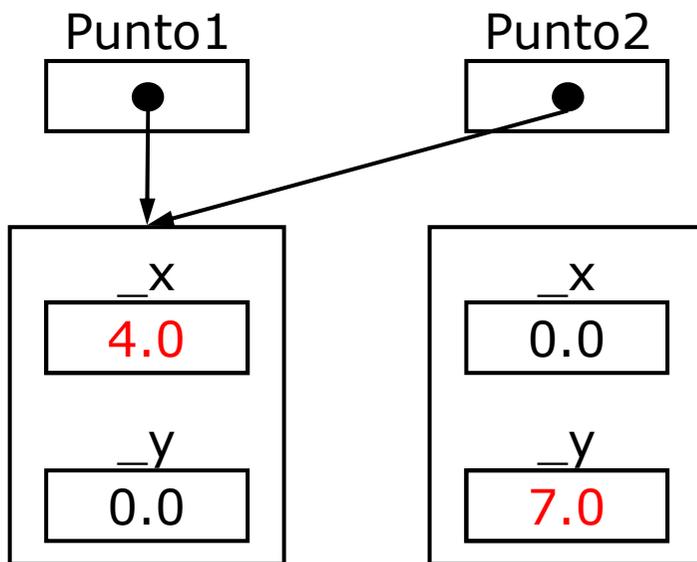
```
Punto2 = Punto1;
```



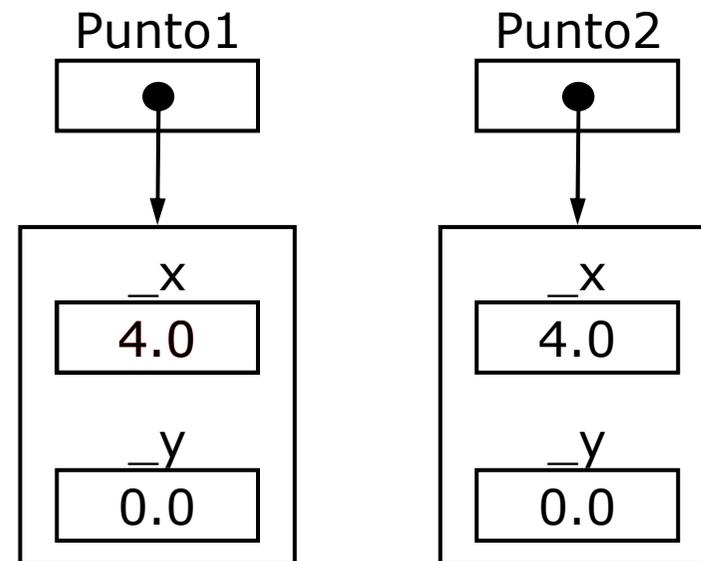
! Este objeto se ha perdido:
Entra en acción el "recolector
de basura" (*garbage collector*)

Clases y objetos en Java (XI)

- Comparación entre objetos:
 - Compara referencias, no valores de atributos



Punto1 == Punto2 → **true**



Punto1 == Punto2 → **false**

Clases y objetos en Java (XII)

- Modificadores de clase:

```
[modificador] class nombre_clase  
{  
    Variables ...  
  
    Métodos ...  
}
```

Clases y objetos en Java (XIII)

- Tipos de clases (*modificador*):
 - **Pública (public)**: accesible desde otras clases (del mismo paquete). Para acceder desde otros paquetes, primero tienen que ser importadas.
 - **Abstracta (abstract)**: *no se instancia*, sino que se utiliza como clase ***base para la herencia***.
 - **Final (final)**: clase que ***termina*** una cadena de herencia. No se puede heredar de una clase final.

- **Sincronizada (synchronized)**: todos los métodos definidos en la clase son sincronizados → no se puede acceder al mismo tiempo a ellos desde distintos threads

Clases y objetos en Java (XIV)

Hola Mundo!!

```
/**
 * Programa HolaMundo
 * que presenta el mensaje Hola Mundo
 */

public class HolaMundo
{
    public static void main (String [] args)
    {
        System.out.println (" Hola Mundo!! ");
    }
}
```

Clases y objetos en Java (XV)

- Características de las clases en Java:
 - Todas las variables y funciones deben pertenecer a una clase → No hay variables ni funciones globales
 - Si una clase deriva de otra **hereda** todas sus variables y métodos
 - Una clase sólo puede heredar de una única clase → “no hay herencia múltiple”
 - Si al definir una clase no se especifica la clase de la que deriva → **por defecto deriva** de la clase **Object** (base de la jerarquía de Java)

Clases y objetos en Java (XVI)

- Características de las clases en Java:
 - ❑ En un fichero pueden existir varias clases pero **sólo una** pública (*public*)
 - ❑ El fichero (.java) debe llamarse como la clase pública

```
class Circulo {  
    ...  
}  
  
public class Prueba  
{  
    public static void main (String [] args)  
    {  
        Circulo c = new Circulo();  
    }  
}
```

Prueba.java

Clases y objetos en Java (XVII)

- **Paquetes** (*packages*):
 - Es una agrupación de clases
 - En la API de Java 1.2 existían 59 paquetes estándar ...
 - El usuario puede crear sus propios paquetes
 - Para que una clase pertenezca a un paquete hay que introducir como *primera sentencia*:
 - ***package nombrePaquete;***
 - El nombre de un paquete puede constar de varios nombres unidos por puntos:
 - Ejemplo: java.awt.event
 - Todas las clases que forman parte de un paquete **deben** estar en el mismo directorio

Clases y objetos en Java (XVIII)

- **Paquetes** (continuación):
 - Se usan con las siguientes finalidades:
 - Agrupar clases relacionadas (java.Math , java.lang, ...)
 - Para evitar conflictos de nombres → el dominio de nombres de Java es Internet
 - Para ayudar en el control de la accesibilidad de clases y miembros (private/public/package)
- Importación de paquetes:
 - Sentencia **import**:
 - ***import nombrePaquete;***
 - Sólo se importa el paquete y no subpaquetes:
 - Ejemplo: Si se importa java.awt no se importa java.awt.event

Clases y objetos en Java (XIX)

- Ejemplos:
 - Importación de una clase:
 - `import es.udc.fic.tp.ordenar.QuickSort;`
 - Importación de **todo** un paquete:
 - `import es.udc.fic.tp.ordenar.*;`
 - En ambos casos en el classpath debe especificarse el directorio del paquete:
 - `CLASSPATH /es/udc/fic/tp/ordenar`

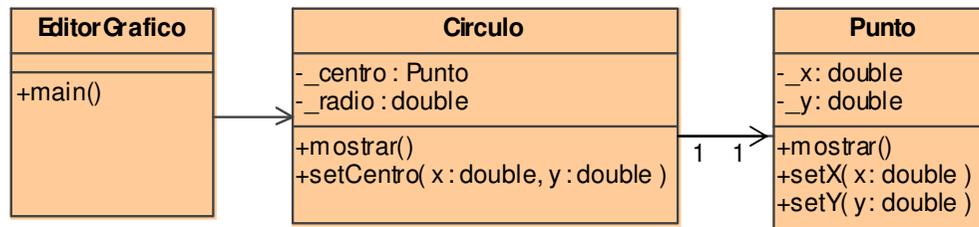
Clases y objetos en Java (XX)

- Control de acceso a las **variables y métodos**:
 - **private**: sólo pueden ser accedidos desde dentro de la clase (no desde las subclases)
 - **protected**: sólo pueden ser accedidos dentro de la **clase**, las **subclases** de la clase y las clases del **paquete**
 - **public**: cualquier clase desde cualquier lugar puede acceder a las variables y métodos
 - **friendly o package** (opción por defecto si no se indica nada): son accesibles por todas las clases dentro del mismo paquete, pero no por los externos al paquete (es como si fuese public, pero sólo dentro del package)

Clases y objetos en Java (XXI)

<i>Especificador</i>	<i>Clase</i>	<i>Subclase</i>	<i>Paquete</i>	<i>Mundo</i>
private	✓			
protected	✓	✓	✓	
public	✓	✓	✓	✓
package	✓		✓	

Clases y objetos en Java (XXII)



Circulo.java

```

package graficos;
public class Circulo {
    private Punto _centro;
    private double _radio;

    public Circulo() //constructor
    {
        _centro = new Punto();
    }

    public void setCentro(double a, double b)
    {
        _centro.x(a);
        _centro.y(b);
    }

    public void setRadio(double d)
    {
        _radio = d;
    }

    public void mostrar()
    {
        System.out.println("Circulo:
        " + _radio + " ");
        _centro.mostrar();
    }
}
  
```

graficos
(paquete)

```

import graficos.*;
public class EditorGrafico
{
    public static void main
    (String [] args)
    {
        Circulo c = new Circulo();
        c.setCentro(3 , 4);
        c.mostrar();
        c.radio(2.5);
        c.mostrar();
    }
}
  
```

Prueba.java

```

package graficos;
class Punto
{
    private double _x;
    private double _y;

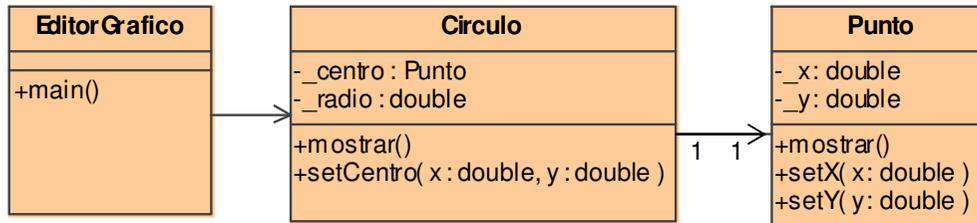
    void setX(double d)
    {
        _x = d;
    }

    void setY(double d)
    {
        _y = d;
    }

    void mostrar()
    {
        System.out.println("Val
        de x e y: " + _x + "
    }
}
  
```

Punto.java

Clases y objetos en Java (XXIII)



Circulo.java

graficos
(paquete)

```

import graficos.*;
public class EditorGrafico
{
    public static void main
    (String [] args)
    {
        Circulo c = new Circulo();
        c.setCentro(3 , 4);
        c.mostrar();
        c.radio(2.5);
        c.mostrar();
    }
}
  
```

Prueba.java

```

package graficos;
class Punto
{
    private double _x;
    private double _y;

    void setX(double d)
    {
        _x = d;
    }
    void setY(double d)
    {
        _y = d;
    }
    void mostrar()
    {
        System.out.println("Valores de x e y: " + _x + " " + _y);
    }
}
  
```

Punto.java

```

package graficos;
public class Circulo {
    private Punto _centro;
    private double _radio;

    public Circulo() //constructor
    {
        _centro = new Punto();
    }

    public void setCentro(double a, double b)
    {
        _centro.setX(a);
        _centro.setY(b);
    }

    public void setRadio(double d)
    {
        _radio = d;
    }

    public void mostrar()
    {
        System.out.println("Circulo: " + _radio + " ");
        _centro.mostrar();
    }
}
  
```

Clases y objetos en Java (XXIV)

■ Clases abstractas:

- Es una clase de la que no se pueden crear objetos
- Utilidad: permitir que otras clases deriven de ella proporcionando un modelo y *métodos generales* de utilidad
- Se declaran empleando la palabra **abstract**:
 - `public abstract class Geometria { . . . }`
- Pueden contener *implementación genérica* de los métodos.

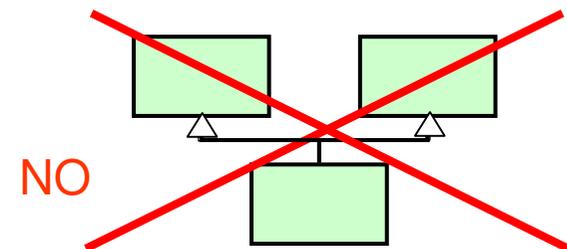
■ Métodos de clase (*static*) ... p.ej main()

- Actúan sobre la clase. No actúan sobre objetos a través del operador punto
- Se definen usando la palabra **static**
- Para usarlos se utiliza el nombre de la clase: `Math.sin(1)`
 - no necesito crear una instancia previamente [new] !!!
- Son lo más parecido a variables y funciones globales de otros lenguajes como, por ejemplo, C/C++

Clases y objetos en Java (XXV)

■ Herencia:

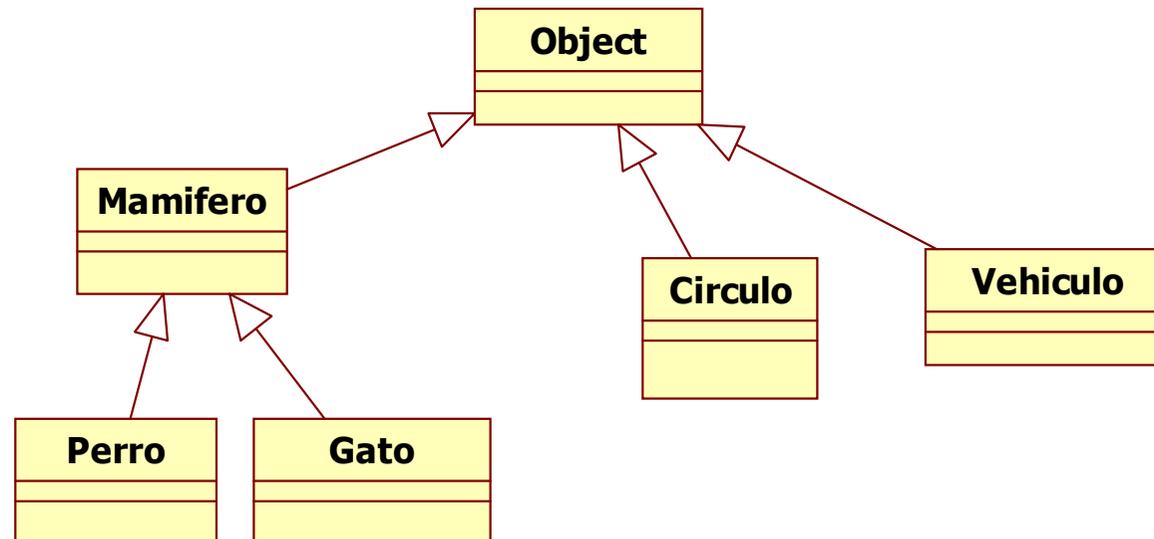
- Construcción de una clase a partir de otra
 - Ejemplo: Mamífero, Perro, Gato
- Para indicar que una clase deriva de otra: **extends**
 - Ejemplo: `class Perro extends Mamifero { ... }`
- Cuando una clase deriva de otra hereda todas sus variables y métodos (implementación general)
 - Pueden ser *redefinidas* en la clase derivada (subclase)
 - Puede añadir nuevas variables y/o métodos
- No se permite herencia múltiple ?:



Clases y objetos en Java (XXVI)

■ Herencia (continuación):

- Todas las clases creadas por el programador tienen una superclase:
 - Cuando no se especifica deriva de `java.lang.Object`
 - La clase `java.lang.Object` es la raíz de toda la jerarquía de clases



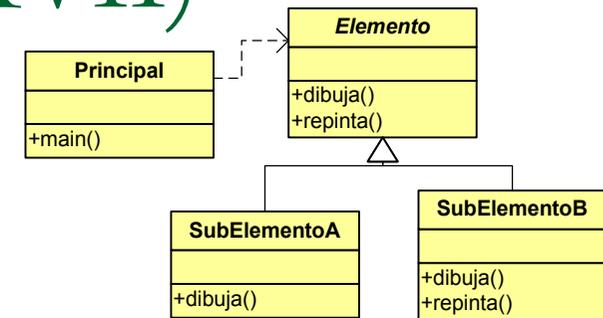
Clases y objetos en Java (XXVII)

Ejemplo1. Herencia simple

```
abstract class Elemento {  
    public abstract void dibuja();  
    public void repinta() {  
        System.out.println("repintando Elemento...");  
    }  
}
```

```
class SubElementoA extends Elemento {  
    public void dibuja(){  
        System.out.println("dibujando SubElementoA...");  
    }  
}
```

```
class SubElementoB extends Elemento {  
    public void dibuja(){  
        System.out.println("dibujando SubElementoB...");  
    }  
    public void repinta(){  
        System.out.println("repintando SubElementoB...");  
    }  
}
```



```
public class Principal {  
    public static void main(String[] args) {  
  
        System.out.println("ejecutando");  
        Elemento A = new SubElementoA();  
        Elemento B = new SubElementoB();  
        A.dibuja();  
        B.dibuja();  
  
    }  
}
```

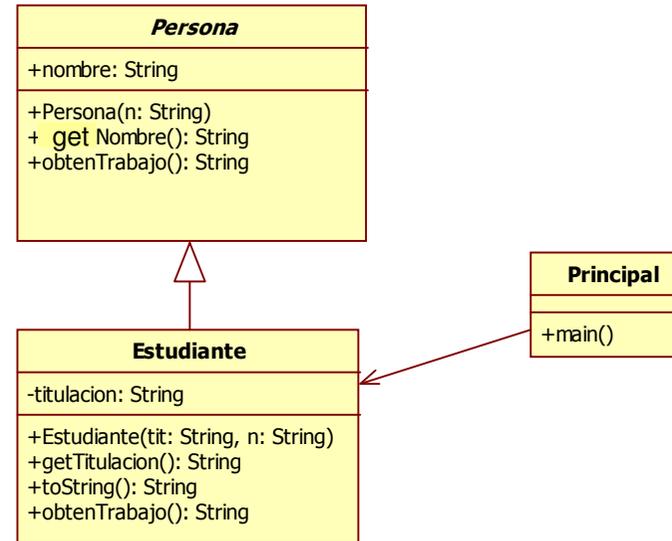
```
$javac -d. Principal.java  
$java Principal  
dibujando SubElementoA...  
dibujando SubElementoB...  
repintando Elemento...  
repintando SubElementoB...
```

Clases y objetos en Java (XXVIII)

Ejemplo2. Herencia simple

```
abstract class Persona {
    private String nombre;
    protected Persona (String n) {
        nombre = n;
    }
    public String getNombre() {
        return nombre;
    }
    abstract public String obtenTrabajo();
}
```

```
class Estudiante extends Persona {
    private String _titulacion;
    public Estudiante (String tit, String n) {
        super(n);
        _titulacion = tit;
    }
    public String getTitulacion() {
        return _titulacion;
    }
    public String toString() {
        return getNombre() + ", " + obtenTrabajo();
    }
    public String obtenTrabajo() {
        return "Estudiante de " + getTitulacion();
    }
}
```



```
public class Principal {
    public static void main (String[] args)
    {
        Estudiante estu =
            new Estudiante("Enx. Informática", "Pepe");
        System.out.println( estu ); // llama a "toString"
        System.out.println( estu.getNombre() );
    }
}
```

```
$ javac -d. Principal.java
$ java Principal
Pepe, Estudiante de Enx. Informática
Pepe
```

Clases y objetos en Java (XXIX)

■ Interfaz (Interface).

- Puede ser vista como una clase abstracta en la que ningún método puede tener implementación.

Ej: interface Cloneable { ... }

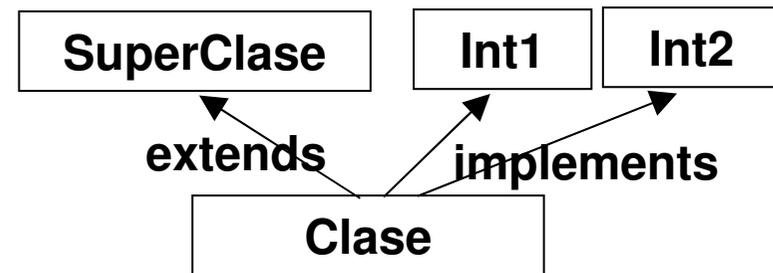
- Métodos son implícitamente públicos y abstractos.
- Atributos son implícitamente: public, static y final.
- Una clase que implemente una interfaz ha de definir todos sus métodos.

- Cláusula **implements**

Ej: class Circulo extends Elemento implements Cloneable

- 1 clase puede:
 - extender 1 superclase
 - Implementar n interfaces.

JAVA: Permite Herencia múltiple por implementación.



Clases y objetos en Java (XXX)

- Ejemplo 1. Uso común de interface

```
interface ConNombre {  
    public String obtenerNombre();  
}
```

```
class UnaClaseConNombre implements  
    ConNombre {  
    public String obtenerNombre() {  
        return "Un nombre";  
    }  
}
```

Ha de implementar **obtenerNombre()**

```
interface ConNombre {  
    public String obtenerNombre();  
}
```

```
interface Usuario extends ConNombre {  
    public boolean autorizar (Usuario u);  
    public int obtenerUID();  
}
```

```
class UsuarioImpl implements Usuario {  
    public String obtenerNombre() {return "Un nombre"; }  
    public boolean autorizar(Usuario u) {return false;}  
    public int obtenerUID() {return -1;}  
}
```

```
    public void metodo1() {.....}  
    public int  metodo2() {.....}
```

```
}
```

Ha de implementar **obtenerNombre()**,
autorizar() y **obtenerUID()**, porque al ser
Usuario un interface, no puede haber
implementación allí.

Clases y objetos en Java (XXXI)

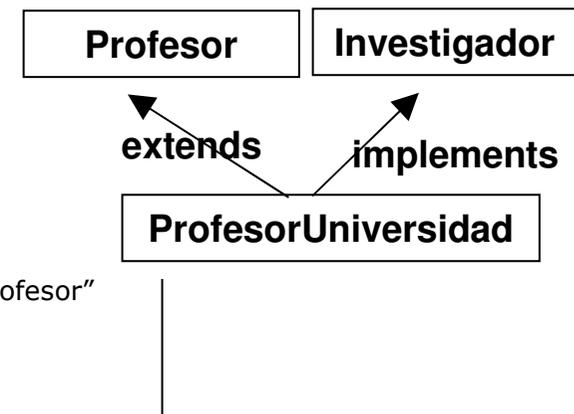
■ Ejemplo 2. Herencia múltiple

```
class Profesor {  
    private String nombre;  
    public Profesor (String n) {  
        this.nombre = n;  
    }  
    public String getNombre() {  
        return nombre;  
    }  
}
```

```
interface Investigador {  
    public String getCampoInvestigacion() ;  
}
```

```
class ProfesorUniversidad extends Profesor implements Investigador {  
    private String investigacion;  
  
    public ProfesorUniversidad(String nom, String inv) {  
        super(nom);  
        this.investigacion = inv;  
    }  
    public String getCampoInvestigacion() {  
        return investigacion;  
    }  
    // Implementa el interface  
  
    //getNombre() se hereda...  
}
```

```
public class Principal {  
    public static void main(String[] args) {  
        Profesor A = new Profesor("Pepe Botella");  
        Profesor B = new ProfesorUniversidad("Nirvana","Musica Celestial");  
        ProfesorUniversidad C = new ProfesorUniversidad("Miguelón","Deportes");  
        System.out.println(A.getNombre());  
        System.out.println(B.getNombre());  
        System.out.println(B.getCampoInvestigacion()); //ERRÓNEO "tratado como Profesor"  
        System.out.println(C.getCampoInvestigacion());  
    }  
}
```



Tipos primitivos (I)

- El tamaño de los tipos no varía entre plataformas

Tipo primitivo	Tamaño	Mínimo	Máximo	Tipo envoltura
boolean	-	-	-	Boolean
char	16 bits	Unicode 0	Unicode $+2^{16} - 1$	Character
byte	8 bits	-128	+127	Byte
short	16 bits	-2^{15}	$+2^{15} - 1$	Short
int	32 bits	-2^{31}	$+2^{31} - 1$	Integer
long	64 bits	-2^{63}	$+2^{63} - 1$	Long
float	32 bits	-3.4×10^{38}	$+3.4 \times 10^{38}$	Float
double	64 bits	-3.4×10^{308}	$+3.4 \times 10^{308}$	Double
void	-	-	-	Void

Tipos primitivos (II)

- Clases envoltura de los tipos primitivos:
 - Se puede declarar un tipo primitivo como no primitivo (manejo como objeto).
 - Ejemplo:

```
char c = 'x';
```

```
Character C = new Character('x');
```

```
c = C.charValue();
```

Literales

- **Numéricos:** como en otros lenguajes
 - Se usa una L para que sea long y no int: 29L
 - Se usa una F para que sea float y no double: 29.3F
- **Carácter:** comillas simples ('a')
 - Caracteres especiales:
 - \n : salto de línea
 - \t : tabulador
 - Etc.
- **Cadenas caracteres (String):** comillas dobles ("Mi cadena")

Variables

- Definición: **tipo nombreVariable;**
- Variables de la clase: se les asigna un valor por defecto (ej: int → 0)
- Variables locales de métodos: deben inicializarse siempre de forma explícita
- Modificador **final**:
 - Impide que se altere el valor de la variable → **constantes**
 - Ejemplos:

```
final double PI = 3.1416;  
final int MAXIMO = 100;
```

Identificadores (I)

- Normas:
 - Comienza por una letra, un guión bajo (_) o un símbolo de dólar (\$)
 - Los demás caracteres pueden ser letras o dígitos
 - Ejemplos:
 - Correctas: midato, _midato, \$midato
 - Incorrectas: 7clases , ?clases
 - Java distingue entre mayúsculas y minúsculas

Identificadores (II)

■ **Convenios: !!**

- ❑ Nombres de clase: empiezan por mayúscula (**C**irculo)
- ❑ Nombres de métodos o atributos: empiezan por minúscula (**p**rintln() , **u**nCirculo, ...)
- ❑ Constantes: todo en mayúsculas (Math.**PI**)
- ❑ Identificadores formados por varias palabras: comienzo de cada nueva palabra en mayúsculas
 - Ejemplos: dibuja**C**uadrado(), Clase**C**uadrado, un**C**uadrado

Operadores (I)

- Se parece a C ...??
 - Aritméticos: +, -, *, /, %
 - Asignación: =, +=, -=, *=, /=, %=
 - Autoincrementales: ++, --
 - Relacionales: >, >=, <, <=, ==, !=
 - Lógicos: &&, ||, !, &, |
 - Concatenación de cadenas: +

Operadores (II)

- Ejemplos:

`d = e++;` // Se asigna **e** a **d** y luego se incrementa **e**

`d = ++e;` // Se incrementa **e** y luego se asigna **e** a **d**

`a += b;` // equivale a `x = x + y;`

`a *= b;` // equivale a `x = x * y;`

`c = 3;`

`a = c++;` // Resultado: `a = 3` y `c = 4`

`a = ++c;` // Resultado: `a = 4` y `c = 4`

`"perro" + "gato"` // Resultado `perrogato`

Comentarios

- `//` comentarios para una sola línea
- `/*` comentarios de una o más líneas `*/`
- *`/**` comentario de documentación, de una o más líneas. Permite generar documentación automáticamente con la herramienta `javadoc` `*/`*

Métodos (I)

- Son similares a las funciones de otros lenguajes
- Definición:

```
[modificadores] tipo nombre (parámetros)
{
    // Cuerpo del método
}
```

- ❑ modificadores: indican el control de acceso al método
- ❑ tipo: tipo del dato que devuelve el método (void = nada)
- ❑ parámetros: declaraciones separadas por comas

Métodos (II)

- Para devolver valores: **return**
- El método termina
 - Al llegar a la llave de cierre ó
 - Al ejecutar el return
- Si el método no es de tipo *void* **debe** terminar siempre con un return
- Si el método es de tipo *void* se puede forzar el fin con la instrucción: return;

Métodos (III)

■ Ejemplo:

```
public int min(int a, int b)
{
    int minimo;
    if (a<b)
        minimo = a;
    else
        minimo = b;
    return minimo;
}
```

También:

```
public int min(int a, int b) {
    return (a<b) ? a : b;
}
```

Cada parámetro con su tipo.
No es válido: int a, b

Indica el valor que
devuelve el método

Métodos (IV)

- Ejemplo:

```
private void mostrar(int numero)
{
    System.out.println("Valor: " + numero);
}
```

Métodos (V)

- Métodos especiales: **constructores**
 - Invocados automáticamente en la creación de un objeto
 - El nombre del constructor es el **mismo que el de la clase**
 - Si **no** se especifica un **constructor** → por defecto (no inicializa nada, sólo reserva memoria)

- Ejemplo:

```
class Ejemplo {  
    int dato;  
    Ejemplo() {  
        System.out.println("Creando instancia de Ejemplo");  
        dato = 10;  
    }  
}
```

Métodos (VI)

- **Sobrecarga** de métodos:

- Varios métodos con el mismo nombre pero diferente firma

- Ejemplo:

```
public class Ejemplo {  
    public int interes(int a, int b) { . . . . . }  
    public int interes(double a, double b) { . . . . . }  
    public int interes(int a, int b, int c) { . . . . . }  
    public int interes() { . . . . . }  
}
```

- Se diferencian por el tipo y número de parámetros
- Java llamará a uno u otro en función de los parámetros pasados durante la "llamada del método"

Métodos (VII): Clonación

- Paso por referencia y valor:
 - Por valor: los tipos primitivos
 - Por referencia: los objetos
- Si se quiere pasar por valor los objetos:
 - Hacer una copia antes de pasarlo: Clonación de objetos
 - Método: **clone()**
 - Indicar que la clase es clonable: **implements Cloneable**
 - Se copian automáticamente todos atributos primitivos
 - Obligación de clonar atributos "de objeto" (sólo copia referencias)

Métodos (VIII): Clonación

Clonacion.java

Ejemplos:

```
public class Referencia
{
    int dato;
    Referencia (int valor) {
        dato = valor;
    }
    public static void main(String[] args)
    {
        Referencia a = new Referencia(10);
        //Referencia al objeto
        Referencia b = a;
        System.out.println("a: " + a.dato);
        System.out.println("b: " + b.dato);
        a.dato++;
        System.out.println("a: " + a.dato);
        System.out.println("b: " + b.dato);
    }
}
```

```
E:\TP\ejem
E:\TP\ejem
a: 10
b: 10
a: 11
b: 11
```

Referencia.java

```
public class Clonacion implements Cloneable
{
    int dato;
    Clonacion (int valor) {
        dato = valor;
    }
    public static void main(String[] args)
        throws CloneNotSupportedException
    {
        Clonacion a = new Clonacion(10);
        // Clona el objeto
        Clonacion b = (Clonacion) a.clone();
        System.out.println("a: " + a.dato);
        System.out.println("b: " + b.dato);
        a.dato++;
        System.out.println("a: " + a.dato);
        System.out.println("b: " + b.dato);
    }
}
```

```
E:\TP\ejemplos>javac Clonacion.java
E:\TP\ejemplos>java -classpath e:\tp\ejemplos Clonacion
a: 10
b: 10
a: 11
b: 10
```

Métodos (IX): Clonación

■ **Object.clone()**

- ❑ *Protected* clone (); → accesible sólo desde jerarquía.
- ❑ Reserva memoria necesaria.
- ❑ Copia bit a bit.
- ❑ Devuelve 1 objeto Object

- ❑ Se usará de base para implementar nuestro "clone()"
 - Normalmente 1ª instrucción = super.clone()

Métodos (X): Clonación

- Clonación Objetos simples.

- Implementar **interface Cloneable**

- Object.clone() chequea si es clonable.

Sino → CloneNotSupportedException

- Reescribir como público. public Object clone() {...}

- Llamar a super.clone() al principio

```
class MiObjeto implements Cloneable {
    private int _i;

    public MiObjeto (int i) {this._i = i; }
    public void incrementa() {_i++;}
    public String toString() {return ""+_i;}

    public Object clone(){ //makes it visible.
        try {
            return super.clone();
        }
        catch (CloneNotSupportedException e) {
            System.err.println("Obj no clonable");
            return null;
        }
    }
}
```

```
Principal {
    public void main(String[] args) {
        m= new MiObjeto(2);
        n = m;
        o = (MiObjeto) m.clone();
        incrementa();
        System.out.println(m.toString() + " " +
            n.toString()+ " " + o.toString());
    }
} //Imprime: 3 3 2
```

Métodos (IX): Clonación

- Clonación Objetos compuestos.
 - 1. Copia superficial.
 - Como en objetos simples.
 - Copia solamente referencias a objetos.
 - 2. Copia en profundidad.
 - Clonar superficialmente el Objeto compuesto.
 - Clonar a su vez cada una de las referencias a los objetos
 - Implica que dichos objetos han de ser clonables.

```

class LeerProfundidad implements Cloneable {
    private int _prof;
    public LeerProfundidad (int p) {this._prof = p; }
    public Object clone(){ //makes it visible.
        Object o = null;
        try {
            o = super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace(System.err);
        }
        return o;
    }
}

```

```

class LeerTemperatura implements Cloneable {
    private int _tmp;
    public LeerTemperatura (int t) {this._tmp = t; }
    public Object clone(){ //makes it visible.
        Object o = null;
        try {
            o = super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace(System.err);
        }
        return o;
    }
}

```

```

class LeerTodo implements Cloneable {
    private LeerProfundidad _prof;
    private LeerTemperatura _temp;
    public LeerTodo (int t, int p) {
        _prof = new LeerProfundidad(p);
        _temp = new LeerTemperatura(t);
    }
    public Object clone(){ //makes it visible.
        LeerTodo o = null;
        try {
            o = (LeerTodo) super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace(System.err);
        }
        o._prof = (LeerProfundidad) _prof.clone();
        o._temp = (LeerTemperatura) _temp.clone();
        return o;
    }
}

```

```

public class principal {
    public static void main(String[] args) {
        LeerTodo miLeer = new LeerTodo(10,20);
        // clonación.
        LeerTodo miLeer2 = (LeerTodo)miLeer.clone();
        System.out.println(miLeer.toString() + " " +
            miLeer2.toString());
    }
}

```

```

class LeerProfundidad implements Cloneable {
    private int _prof;
    public LeerProfundidad (int p) {this._prof = p; }
    public Object clone(){ //makes it visible.
        Object o = null;
        try {
            o = super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace(System.err);
        }
        return o;
    }
}

```

```

class LeerTemperatura implements Cloneable {
    private int _tmp;
    public LeerTemperatura (int t) {this._tmp = t; }
    public Object clone(){ //makes it visible.
        Object o = null;
        try {
            o = super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace(System.err);
        }
        return o;
    }
}

```

```

class LeerTodo implements Cloneable {
    private LeerProfundidad _prof;
    private LeerTemperatura _temp;
    public LeerTodo (int t, int p) {
        _prof = new LeerProfundidad(p);
        _temp = new LeerTemperatura(t);
    }
    public Object clone(){ //makes it visible.
        LeerTodo o = null;
        try {
            o = (LeerTodo) super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace(System.err);
        }
        o._prof = (LeerProfundidad) _prof.clone();
        o._temp = (LeerTemperatura) _temp.clone();
        return o;
    }
}

```

```

public class principal {
    public static void main(String[] args) {
        LeerTodo miLeer = new LeerTodo(10,20);
        // clonación.
        LeerTodo miLeer2 = (LeerTodo)miLeer.clone();
        System.out.println(miLeer.toString() + " " +
            miLeer2.toString());
    }
}

```

Estructuras de control: sentencias condicionales (I)

- Condición simple: **if**

```
if (expresión)  
{  
    sentencia1;  
    . . .  
    sentencia N;  
}
```

- Las llaves delimitan el bloque de sentencias y no son necesarias si sólo hay una sentencia

Estructuras de control: sentencias condicionales (II)

- Condición doble: **if else**

```
if (expresión)
{
    Grupo de sentencias1;
}
else
{
    Grupo de sentencias2;
}
```

Estructuras de control: sentencias condicionales (III)

- Ejemplos:

```
if (calificacion >= 5)
    System.out.println("Aprobado");
else
    System.out.println("Suspenso");
```

```
if (saldo >= importe)
{
    saldo = saldo - importe;
    System.out.println("Saldo suficiente");
}
else
    System.out.println("Error: saldo insuficiente");
```

Estructuras de control: sentencias condicionales (IV)

- Condiciones múltiples: **if else if else ...**

```
if (expresión1){
    Grupo sentencias1;
} else if (expresión2) {
    Grupo sentencias2;
} else if (expresión3) {
    Grupo sentencias3;
} else {
    Grupo sentencias4;
}
```

Estructuras de control: sentencias condicionales (V)

- Condiciones múltiples: **switch**

```
switch (expresión) {  
    case valor1: {Grupo sentencias1; [break];}  
    case valor2: {Grupo sentencias2; [break];}  
    . . .  
    case valorN: {Grupo sentenciasN; [break];}  
                [default: Grupo sentenciasN+1;]  
}
```

Estructuras de control: sentencias condicionales (VI)

- Características del switch:
 - Cada sentencia case se corresponde con un único valor de la expresión → **No** rangos ni condiciones
 - La sentencia default es opcional y sólo se ejecuta si no se cumple ninguno de los case
 - Cuando se ejecuta una sentencia case también se ejecutan las que vienen a continuación → **break**

Estructuras de control: sentencias condicionales (VII)

- Ejemplo:

```
switch (dia) {  
    case 1: System.out.println("Lunes"); break;  
    case 2: System.out.println("Martes"); break;  
    case 3: System.out.println("Miércoles"); break;  
    case 4: System.out.println("Jueves"); break;  
    case 5: System.out.println("Viernes"); break;  
    case 6: System.out.println("Sábado"); break;  
    case 7: System.out.println("Domingo"); break;  
}
```

Estructuras de control: sentencias condicionales (VIII)

- Ejemplo: Número de días del mes 1,2,3...

```
switch (mes) {  
    case 1:  
    case 3:  
    case 5:  
    case 7:  
    case 8:  
    case 10:  
    case 12: NDias = 31; break;  
    case 4:  
    case 6:  
    case 9:  
    case 11: NDias = 30; break;  
    case 2:  
        if ( ((year % 4 == 0) && !(year % 100 == 0)) || (year % 400 == 0) )  
            NDias = 29;  
        else NDias = 28;  
        break;  
}
```

Estructuras de control: sentencias repetitivas (I)

- Bucle **while**:

```
while (expresión) {  
    Grupo de sentencias;  
}
```

- Bucle **do while**:

```
do {  
    Grupo de sentencias;  
} while (expresión)
```

Estructuras de control: sentencias repetitivas (II)

- Ejemplos:

```
contador = 1;
while (contador <= 10)
{
    suma = suma + contador;
    contador++;
}
```

```
contador = 1;
do
{
    suma = suma + contador;
    contador++;
} while (contador <= 10)
```

Estructuras de control: sentencias repetitivas (III)

- Bucle **for**:

```
for (inicialización; expresión; incremento)
{
    Grupo de sentencias;
}
```

Estructuras de control: sentencias repetitivas (IV)

- Ejemplos:

```
for (contador=1;contador<=10;contador++)  
    suma = suma + contador;
```

```
for (i = 10, j = 0; i > j; j++, i--)  
    System.out.println("Una iteración del bucle " + j);
```

Estructuras de control: sentencias repetitivas (V)

- Sentencia **break**:
 - Válida para sentencias condicionales y repetitivas
 - Finaliza la ejecución del bucle (no ejecuta las sentencias que vienen después)
- Sentencia **continue**:
 - Se usa sólo en las sentencias repetitivas
 - Finaliza la iteración actual del bucle y comienza la siguiente iteración

Estructuras de control: sentencias repetitivas (VI)

- Ejemplos:

```
for (suma=0, numero=1; numero<5; numero++)  
{  
    if (numero<=1)  
        break; //sale del bucle  
    suma = suma + numero;  
}
```

suma
0

```
for (suma=0, numero=1; numero<5; numero++)  
{  
    if (numero<=1)  
        continue;  
    suma = suma + numero;  
}
```

suma
2+3+4 = 9

Cadenas de caracteres (I)

- Clase de la biblioteca estándar: **String**
- Las cadenas almacenadas en la clase string no se pueden modificar
 - Son objetos constantes que contienen la cadena que se les asignó durante su creación
- Se pueden crear como cualquier otro objeto:
 - `String cadena = new String(cad);`
- Ejemplos:
 - `String frase = new String("Mi primera cadena");`
 - `String frase = "Mi primera cadena";`

Cadenas de caracteres (II)

- Concatenación de cadenas: operador **+**
 - Ejemplo: `String frase3 = frase2 + frase1;`
- Longitud de la cadena: método **length()**
 - Ejemplo: `int longitud = frase.length(); "abc".length()`
- Acceso a un carácter: método **charAt(índice)**
 - Si la cadena tiene longitud n el índice va de 0 a $n-1$
 - Ejemplo: `char c = frase.charAt(3);`
- Comparación de cadenas: método **equals(String)**
 - Ejemplo: `if (frase1.equals(frase2))`
 - Si se compara `frase1==frase2` se comprueba si son el mismo objeto pero no si contienen la misma cadena !!

```
System.out.println ("LOCO".charAt(2) + " " + "LOCO".length()+  
" " + "a".equals("a") + " " + ("a"=="b")); //→ C 4 true false
```

Cadenas de caracteres (III)

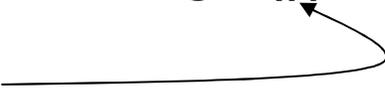
- Subcadenas: método **substring**(*índice1*, *índice2*)
 - **índice1** indica la posición del primer elemento de la subcadena e **índice2-1** el del último elemento
 - *índice1* e *índice2* son enteros (int)
 - Si sólo se le pasa un índice indica el comienzo
 - Ejemplo:

```
String frase = "Mi cadena";
```

```
String subfrase1 = frase.substring(1,5);
```

```
String subfrase2 = frase.substring(4, frase.length());
```

```
String subfrase2 = frase.substring(4);
```

 **equivale**

```
System.out.println(subfrase1); // Muestra: i ca
```

```
System.out.println(subfrase2); // Muestra: adena
```

Arrays (I)

- Clase de la biblioteca estándar: **Array**
- Para **declarar** un array:
 - Especificar el tipo o clase de los elementos
 - Corchetes detrás del tipo o de la variable
 - Ejemplos:
double numeros[];
double[] numeros;
Alumno[] clase;
Empleado empresa[];
- Son colecciones homogéneas de objetos

Arrays (II)

- **Creación** de un array:

- Como todo objeto: con el operador new
- Se indica la longitud entre corchetes después del tipo o clase

- Ejemplos:

```
numeros = new double[30];
```

```
clase = new Alumno[50];
```

```
Empleado empresa[] = new Empleado[60];
```

- **Acceso** a los elementos:

- Con un índice entero entre corchetes a continuación del nombre del array

- Ejemplos: numeros[15] clase[indice]

Arrays (III)

- Si el array tiene longitud n : la primera posición es la 0 y la última la $n-1$
- Se permite la creación de arrays *dinámicos*: determinación del tamaño en tiempo de ejecución
- Es obligación del programador la de controlar que el índice es una posición válida:
 - Si no es así se produce una excepción
- Los arrays disponen de un **atributo público** denominado **length** que contiene el número de posiciones del array
 - Ejemplo:

```
int longitud = numeros.length; // NO ES UN MÉTODO
```

Arrays (IV)

- Arrays de objetos:

- La creación del array no crea ningún objeto de la clase del array
- Deben ser creados los objetos a medida que se usan
- Ejemplo:

```
Persona grupo[];
```

```
grupo = new Persona[100]; /* array listo para  
                               asignarle personas */
```

¡No se crean los 100 elementos del grupo!

```
for (i=0;i<grupo.length;i++) grupo[i] = new Persona();  
System.out.println(grupo[45].toString());
```

Arrays (V)

- Arrays multidimensionales:

 - `tipo[][] nombre = new tipo[tam1][tam2];`

 - `tipo[][][] nombre = new tipo[tam1][tam2][tam3];`

 - ...

 - Ejemplos:

 - `int[][] tabla = new int[5][5];`

 - `Persona[][] grupo = new Persona[10][10];`

Arrays (VI)

- Ejemplo (inicialización –estática- y uso de arrays):

```
public class Ejemplo
{
    public static void main(String [] args)
    {
        int suma, i, numeros[] = {1, 2, 3, 4, 5};

        for (i=0, suma=0 ; i<5; i++)
            suma += numeros[i];

        System.out.println("La suma es: " + suma);
    }
}
```

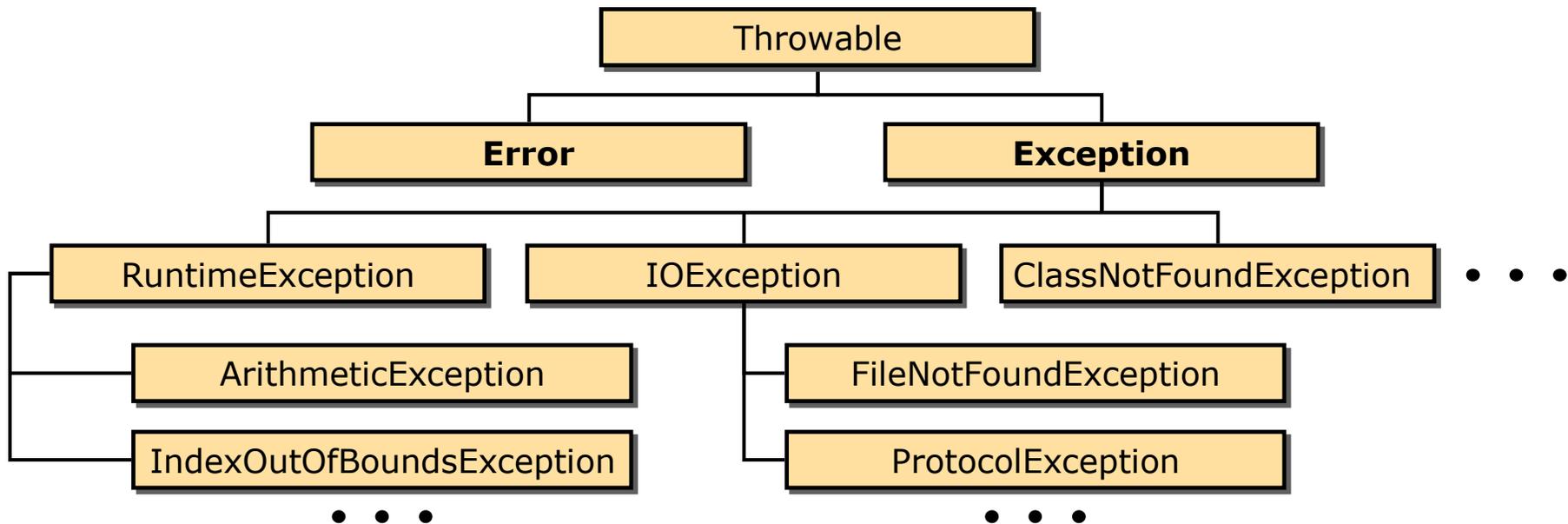
Arrays (VII)

- Ejemplo (array dinámico):

```
public class Ejemplo {
    public static void crear(int longitud) {
        int [] conjunto = new int[longitud];
        System.out.println("Longitud: " + conjunto.length);
    }
    public static void main(String [] args) {
        crear(10);
        crear(20);
    }
}
```

Excepciones (I)

- Excepción: error o condición anormal que se produce durante la ejecución de un programa
- Java permite el manejo o gestión de las excepciones
- Excepciones estándar de Java:



Excepciones (II)

- La clase **Error**:
 - Errores de compilación, del sistema, de la JVM, etc.
 - Son situaciones anómalas e irre recuperables
- La clase **Exception**:
 - Excepciones implícitas:
 - Las de la clase **RuntimeException**
 - Suelen estar producidas por errores de programación
 - Excepciones explícitas:
 - El resto de clases derivadas de *Exception*
 - Java obliga a tenerlas en cuenta y chequear si se producen

Excepciones (III)

- Las clases derivadas de `Exception` pertenecen a distintos `packages`: `java.lang`, `java.io`, etc.
- Pero todas ellas por heredar de *Throwable* pueden usar los métodos:
 - `String getMessage()`: Mensaje asociado a la excepción
 - `String toString()`: Devuelve un `String` que describe la excepción
 - `void printStackTrace()`: Indica el método donde se lanzó la excepción

Excepciones (IV)

- Captura de una excepción:

- Estructura try ... catch ... finally

```
try {  
    // Código que puede producir una excepción  
}  
catch (TipoExcepción excep) {  
    // Gestor de la excepción  
}  
[finally {  
    /* Código que se ejecuta siempre (con excepción o sin  
    ella) */  
} ]
```

Excepciones (V)

- ❑ Si en el código dentro del bloque **try** se produce una excepción de tipo **TipoExcepción** (o descendiente)
 - Se omite la ejecución del resto del código en el bloque **try**
 - Se ejecuta el código situado en el bloque **catch** (gestor)
- ❑ Pueden controlarse diversos tipos de excepciones con varias cláusulas **catch**
 - Se comprobará en el mismo orden que el indicado
 - Sólo se ejecuta 1 bloque catch

Excepciones (VI)

- *Ejemplo:*

```
public class EjemploCatch {
    String mensajes[] = {"Luis", "Carlos", "David" };
    public static void main(String[] args)
    {
        int cont;
        try {
            for (cont = 0; cont <= 3; cont++)
                System.out.println(mensajes[cont]);
        }
        catch (ArrayIndexOutOfBoundsException excep) {
            System.out.println("El array se ha desbordado");
        }
    }
}
```

Excepciones (VII)

- **Relanzar una excepción**

- En ocasiones no interesa gestionar la excepción
- Java permite que el método relance o pase la excepción al método desde el que ha sido llamado
- Cláusula **throws**:

- *Ejemplo:*

```
void metodoEjem() throws IOException, ArithmeticException
{
    // Código que puede lanzar las excepciones
    // No es necesario hacer try ... catch aquí (aunque es posible relanzar)
}
```

- Por tanto hay dos posibilidades:

- Capturar las posibles excepciones y gestionarlas
- Desentenderse de las excepciones y remitirlas al método anterior

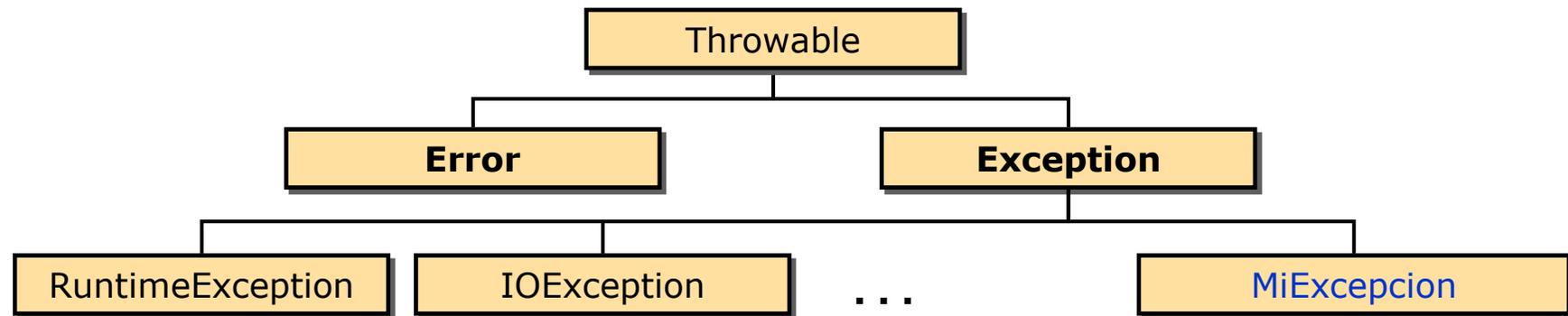
Excepciones (VIII)

- **Crear una excepción propia**

- Extender la clase Exception (u otra excepción)
- Java permite que el método relance o pase la excepción al método desde el que ha sido llamado
- Cláusula **throws**:

- *Ejemplo:*

```
class MiExcepcion extends Exception{  
    public MiExcepcion() {}  
    public MiExcepcion(String message) {super(message);} //para e.getMessage()  
}
```



Entrada/salida estándar (I)

- Regulada a través de la clase **System** del paquete java.lang
 - Contiene, entre otros, 3 objetos:
 - **System.in** : Objeto de InputStream
 - **System.out** : Objeto de PrintStream
 - **System.err** : Objeto de PrintStream
 - Métodos de System.in
 - int read() : lee un carácter y lo devuelve como int
 - Métodos de System.out y System.err
 - int print(cualquier tipo)
 - int println(cualquier tipo)

Entrada/salida estándar (II)

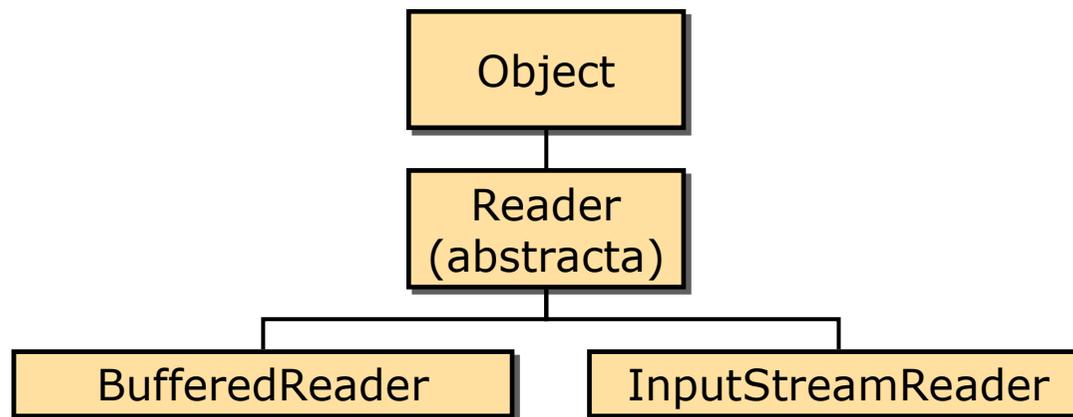
- *Ejemplo:* leer un carácter de teclado.

```
import java.io.*;
public class Ejemplo
{
    public static void main(String [] args) throws IOException
    {
        char caracter;

        caracter = (char) System.in.read();
        System.out.println("Dato leído: " + caracter + ". ");
    }
}
```

Entrada/salida estándar (III)

- Lectura de una línea: Clase **BufferedReader**
 - El método **String readLine()** lee todos los caracteres hasta un `\n`
 - **BufferedReader** necesita un **Reader** en el constructor pero `System.in` es un objeto de la clase **InputStream**:
 - Es necesario usar previamente: **InputStreamReader**



Entrada/salida estándar (IV)

- *Ejemplo (lectura de una línea):*

```
import java.io.*;
public class Ejemplo {
    public static void main(String [] args) throws IOException {
        InputStreamReader canalEntrada = new
            InputStreamReader(System.in);
        BufferedReader entrada = new BufferedReader(canalEntrada);
        String datos;

        datos = entrada.readLine();
        System.out.println("Datos leídos: " + datos);
    }
}
```

Entrada/salida estándar (V)

- *Ejemplo (lectura de un entero):*

```
import java.io.*;
public class Ejemplo {
    public static void main(String [] args) throws IOException {
        InputStreamReader stdin = new InputStreamReader(System.in);
        BufferedReader consola = new BufferedReader(stdin);
        int valor;
        String cadena;

        System.out.println("Introduzca un número: ");
        cadena = consola.readLine();
        valor = Integer.parseInt(cadena);
    }
}
```

Otra clase estándar: Math

- Proporciona dos constantes: **Math.E** y **Math.PI**
- Pertenece a la clase `java.lang`: se importa automáticamente
- Algunos métodos:

`Math.sqrt(num)`

`Math.exp(num)`

`Math.min(a,b)`

`Math.pow(a,b)`

`Math.random()`

`Math.log(num)`

`Math.abs(num)`

`Math.cos(num)`

`Math.sin(num)`

`Math.tan(num)`

`Math.toDegrees(num)`

`Math.toRadians(num)`