

# Introducción a Java

# Contenido

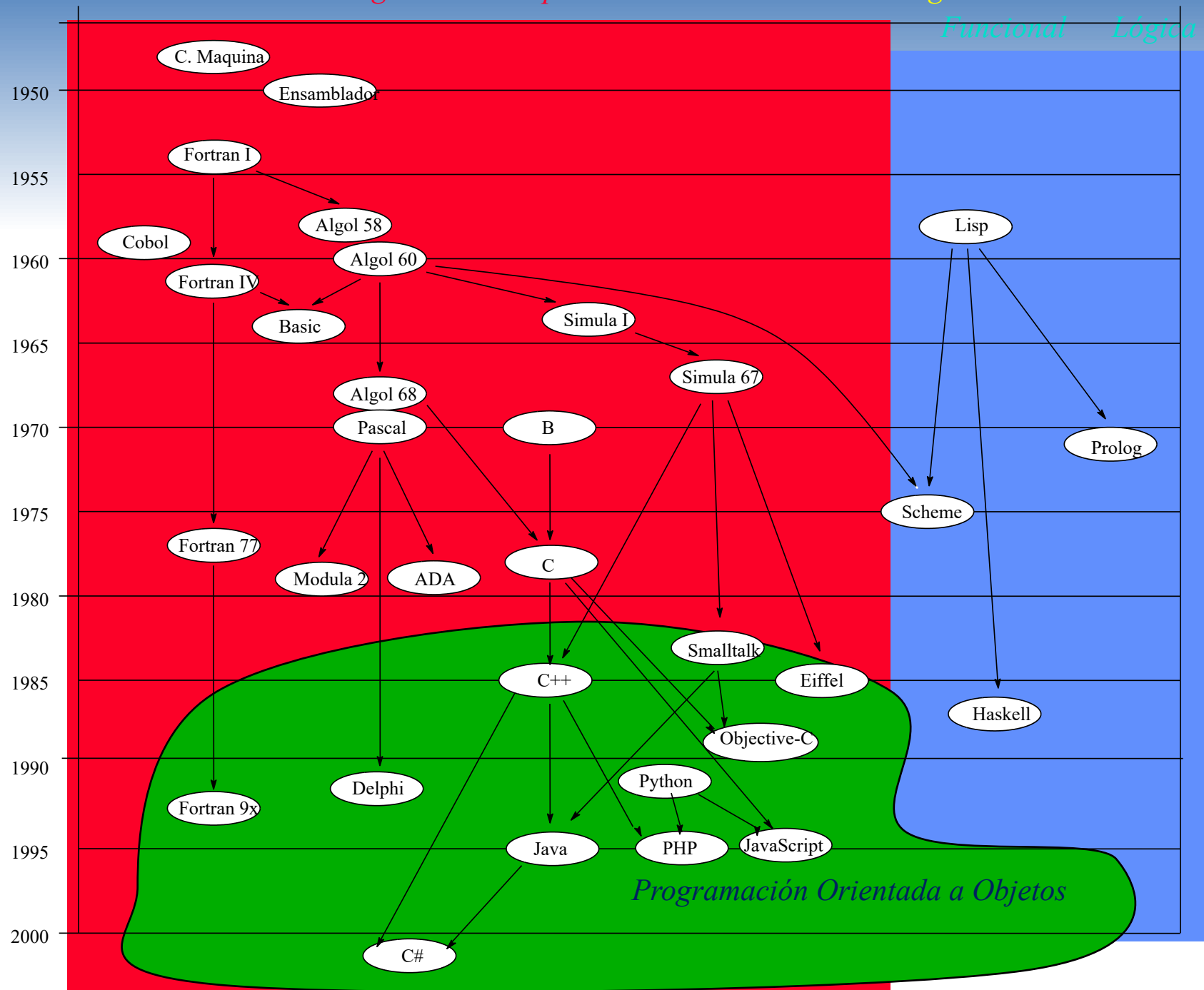
- Introducción histórica
- Programas y paquetes
- Clases y objetos
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores. Excepciones
- Cadenas de caracteres
- Arrays
- Listas
- Herencia y redefinición de comportamiento
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Programación Imperativa

## Programación Declarativa

## Funcional

*Lógica*

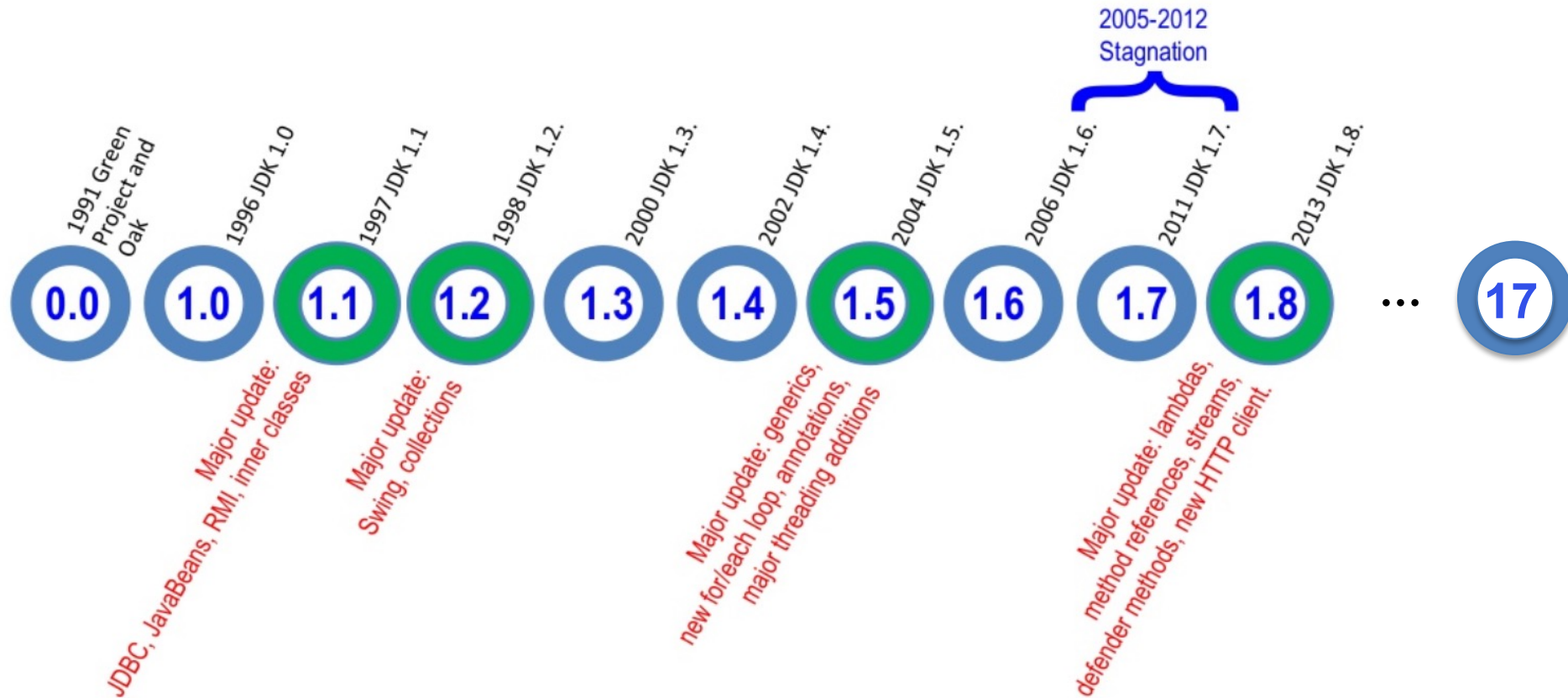


# Introducción a Java

- Desarrollado por Sun. Aparece en 1995
- Basado en C++ eliminando
  - definiciones de tipos de valores y macros,
  - punteros y aritmética de punteros,
  - necesidad de liberar memoria.
- Fiable y seguro:
  - memoria dinámica automática (no punteros explícitos)
  - comprobación automática de tamaño de variables
- Orientado a objetos con:
  - herencia simple y polimorfismo de datos,
  - redefinición de métodos y vinculación dinámica.
  - concurrencia integrada en el lenguaje
  - interfaz gráfica integrada en el lenguaje
- Compilado “especial” (precompilación)
  - ficheros fuente `.java` se convierten en ficheros *bytecode* `.class`
- Interpretado
  - ficheros `.class` son interpretados por la máquina virtual de Java (JVM)

# Java SE Version History

(Green: Major; Blue: Minor)



# Contenido

- Introducción histórica
- **Programas y paquetes**
- Clases y objetos
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores. Excepciones
- Cadenas de caracteres
- Arrays
- Listas
- Herencia y redefinición de comportamiento
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Programa en Java

- Formado por una o varias clases diseñadas para resolver un determinado problema.
- Existe una clase (pública) “distinguida” que contiene un método de clase especial:  

```
public static void main(String[] args)
```

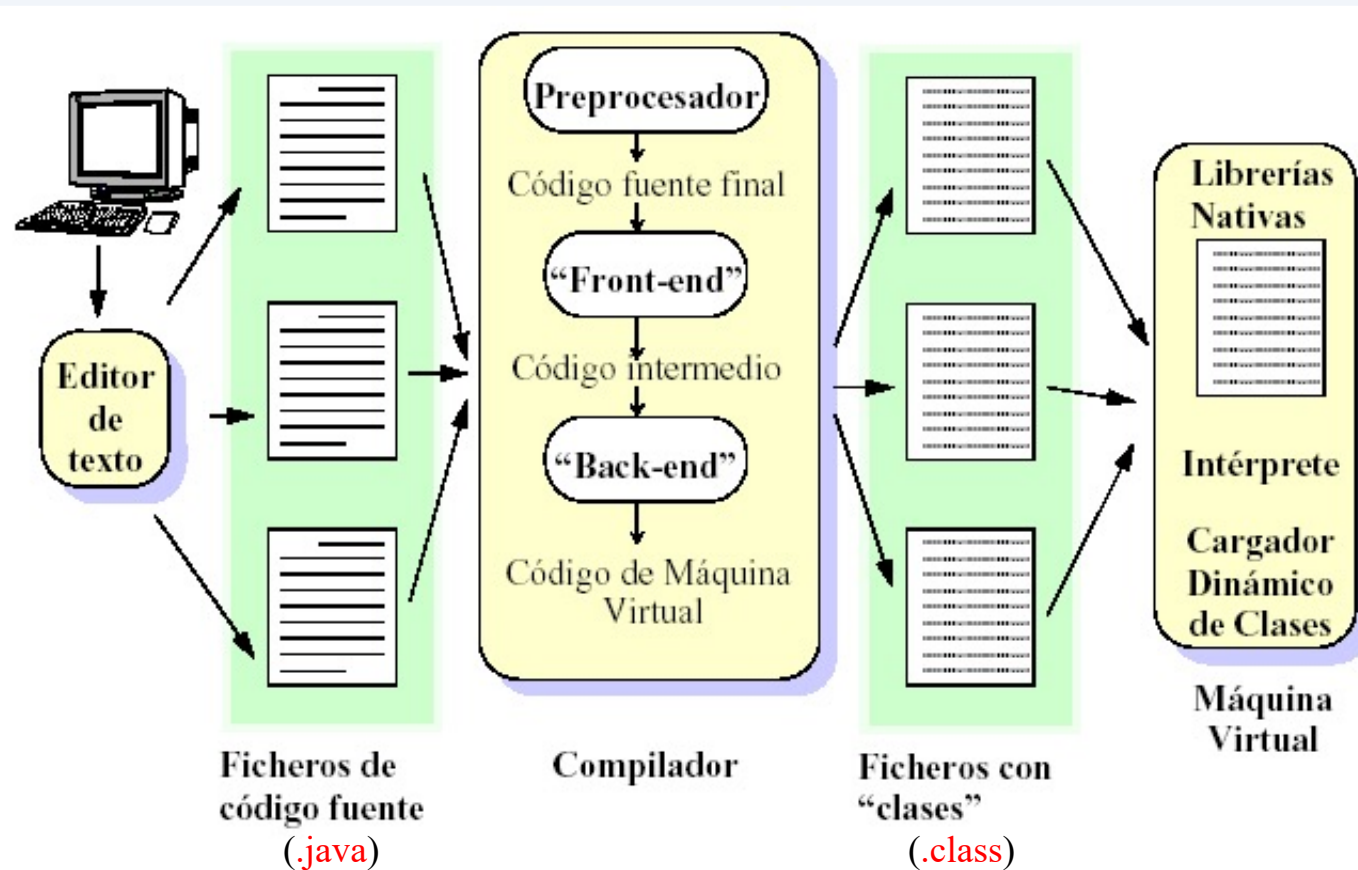
que desencadena la ejecución del programa.
- Esta clase distinguida puede contener más métodos.
- Las demás clases pueden estar definidas *ad hoc* o pertenecer a una biblioteca de clases.
- Cuando se trabaja con un IDE (como Eclipse o IntelliJ), normalmente se crea un proyecto para alojar el programa.

# Ficheros en Java

- Cada **clase declarada como pública** debe de estar en un fichero **.java** con su mismo nombre.
- Cada fichero **.java** puede contener varias clases **pero sólo una podrá ser pública**.
- Cada fichero **.java** debe precompilarse generando un fichero **.class** (en *bytecodes*) por cada clase contenida en él.
- El programa se ejecuta pasando el fichero **.class** de la clase distinguida al intérprete (máquina virtual de Java)

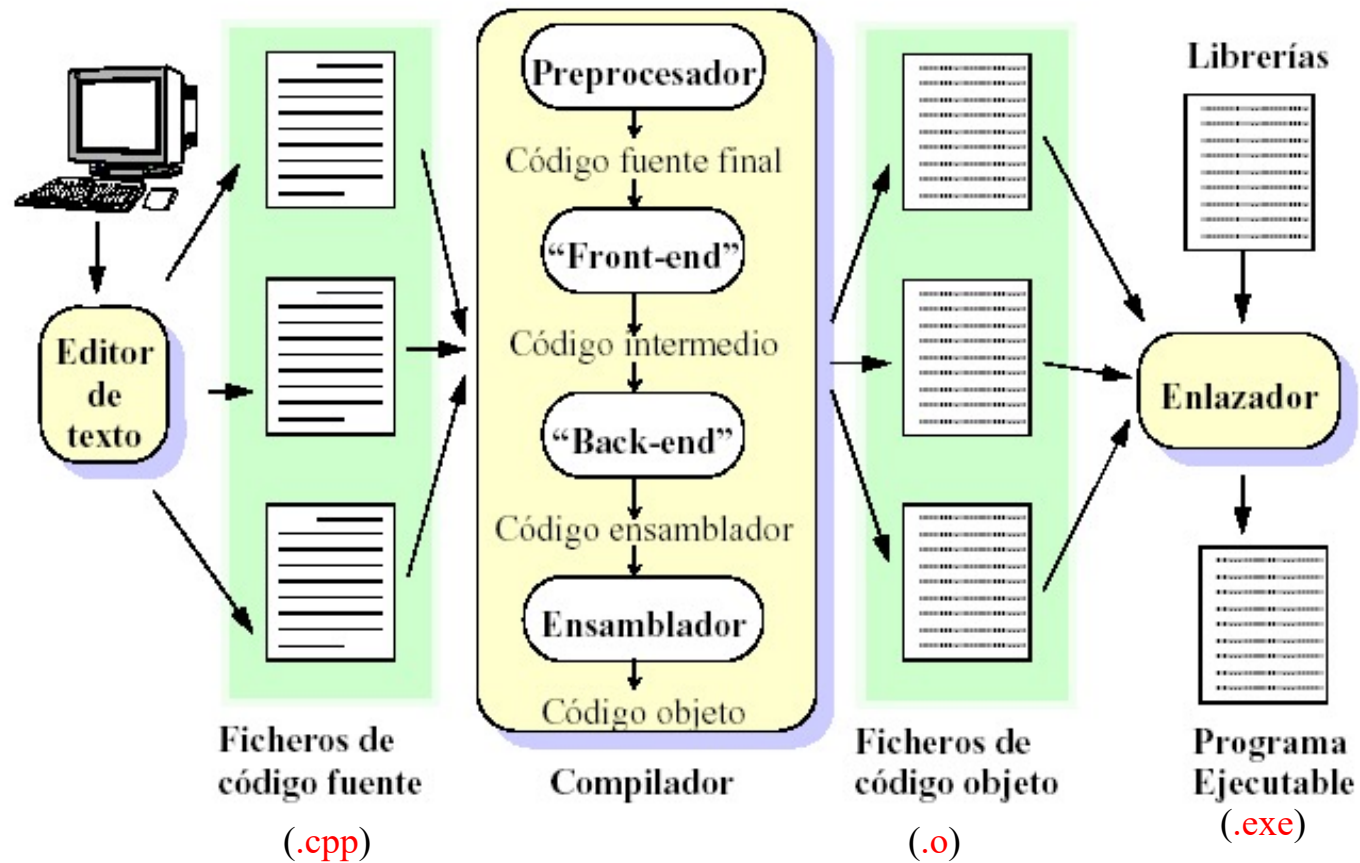


# Compilación e Interpretación en Java



## Compilación e Interpretación (Java)

# Diferencia con otros lenguajes



Compilación y Enlazado (Ej. C++)

# Ejemplo1 (Clase Distinguida)

“Hola.java”

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

# Ejecución de un programa

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

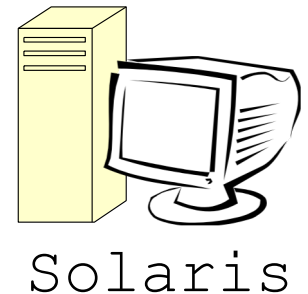
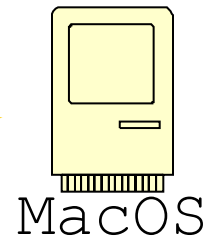
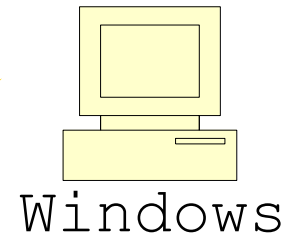
HolaMundo.java

**javac**

Bytecodes

HolaMundo.class

**java**



```
[Mac-Ernesto:Java ernesto$ ls  
HolaMundo.java  
[Mac-Ernesto:Java ernesto$ javac HolaMundo.java  
[Mac-Ernesto:Java ernesto$ ls  
HolaMundo.class HolaMundo.java  
[Mac-Ernesto:Java ernesto$ java HolaMundo  
Hola Mundo  
[Mac-Ernesto:Java ernesto$
```

# Ejemplo2 (Clase Punto)

“Punto.java”

```
public class Punto {  
    private double x, y;  
    public Punto() { this(0,0); }  
    public Punto(double a, double b) { x = a; y = b; }  
    public double abscisa() {return x;}  
    public double ordenada() {return y;}  
    public void abscisa(double a){ x = a; }  
    public void ordenada(double b){ y = b; }  
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }  
    public double distancia(Punto pto) {  
        return Math.sqrt(Math.pow(x - pto.x, 2) +  
            Math.pow(y - pto.y, 2));  
    }  
}
```

# Ejemplo2 (Clase Distinguida)

“TestPunto.java”

```
public class TestPunto {  
    public static void main(String[] args) {  
        Punto p1 = new Punto(1,1);  
        Punto p2 = new Punto(0,0);  
        System.out.println("La distancia entre los puntos es: "  
                           + p1.distancia(p2));  
        System.out.println("Trasladamos el primer punto (+2,+3)");  
        p1.trasladar(2, 3);  
        System.out.println("Ahora La distancia entre los puntos es: "  
                           + p1.distancia(p2));  
    }  
}
```

# Paquetes

- Las bibliotecas se organizan en ***paquetes*** (**package**): agrupación de clases relacionadas.
- Todas las clases de un paquete deben estar localizadas en un mismo directorio.
- A las clases de un paquete se puede acceder directamente con su nombre, solo desde otras clases del mismo paquete.
- Para acceder a una clase desde otro paquete hay que preceder su nombre con el nombre del paquete, o utilizando la cláusula **import**

# Paquetes

```
package paq1;
```

```
public class Punto {  
    private double x, y;
```

```
    public Punto() { this(0,0); }
```

```
    public Punto(double a, double b) { x = a; y = b; }
```

```
    public double abscisa() { return x; }
```

```
    public double ordenada() { return y; }
```

```
    ...
```

```
}
```

Punto.java



# Paquetes

`package paq1;` — mismo paquete

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(double x1, double y1, double x2, double y2) {  
        origen = new Punto(x1, y1);  
        extremo = new Punto(x2, y2);  
    }  
    ...  
}
```

correcto

Segmento.java

# Paquetes

`package paq2;` — **distinto paquete**

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(double x1, double y1, double x2, double y2) {  
        origen = new Punto(x1, y1);  
        extremo = new Punto(x2, y2);  
    }  
    ...  
}
```

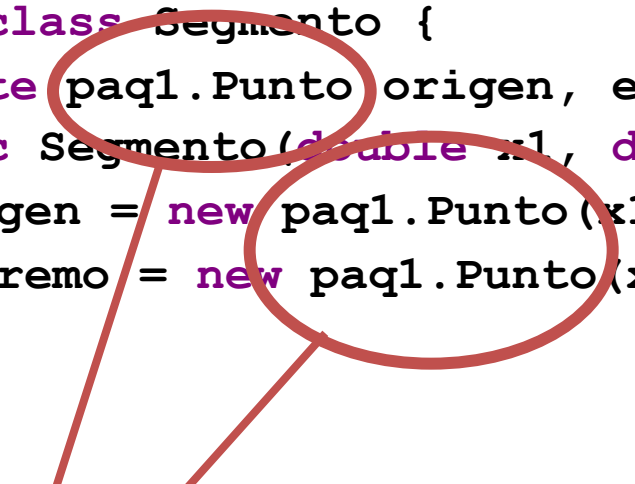
**incorrecto**

Segmento.java

# Paquetes

```
package paq2;
```

```
public class Segmento {  
    private paq1.Punto origen, extremo;  
    public Segmento(double x1, double y1, double x2, double y2) {  
        origen = new paq1.Punto(x1, y1);  
        extremo = new paq1.Punto(x2, y2);  
    }  
    ...  
}
```



**correcto**

Segmento.java

# Paquetes

```
package paq2;
```

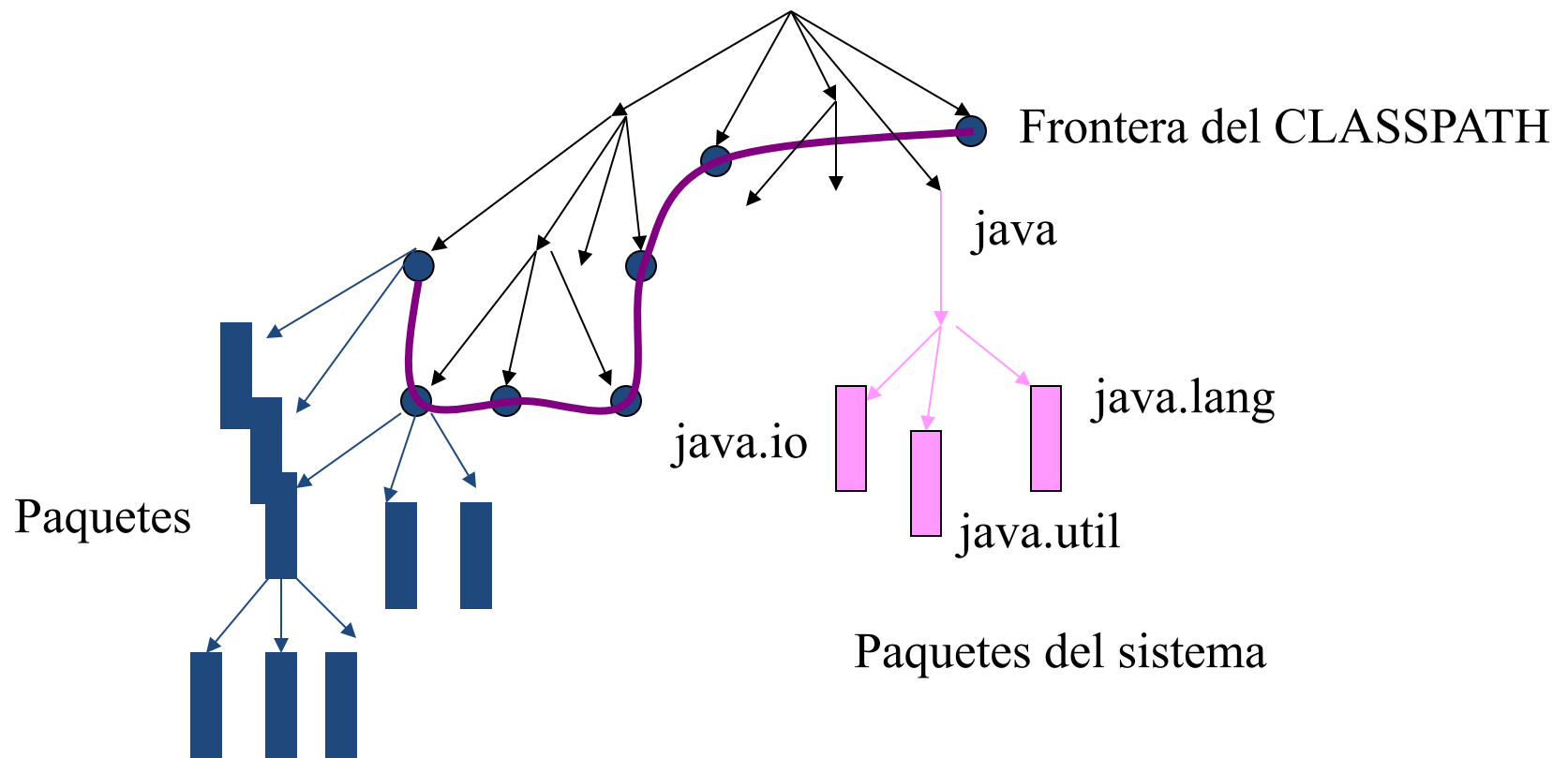
```
import paq1.Punto; // import paq1.*;
```

```
public class Segmento {  
    private Punto origen, extremo;  
    public Segmento(double x1, double y1, double x2, double y2) {  
        origen = new Punto(x1, y1);  
        extremo = new Punto(x2, y2);  
    }  
    ...  
}
```

correcto

Segmento.java

# Estructura de las bibliotecas en Java



# Paquetes básicos del sistema

- `java.lang`: para funciones del lenguaje
- `java.util`: para utilidades adicionales
- `java.io`: para entrada y salida
- `java.text`: para formato especializado
- `java.awt`: para diseño gráfico e interaz de usuario
- `java.awt.event`: para gestionar eventos
- `javax.swing`: nuevo diseño de GUI
- `java.net`: para comunicaciones
- ...

# Acceso a las bibliotecas de Java

- El nombre de cada paquete debe coincidir con el camino que va desde algún directorio del **CLASSPATH** (o desde **/java** o **/javax**) al subdirectorio correspondiente al paquete.
- A las clases incluidas en **java.lang** se puede acceder directamente por sus nombres, p.e.: **System** o **Math**.

# Ejemplo

- Programa para calcular el valor medio de un millón de números generados aleatoriamente, usando las clases
  - `Random` del paquete `java.util`
  - `System` del paquete `java.lang`

```
public class TestAleatorio {  
    public static void main(String[] args) {  
        java.util.Random rnd = new java.util.Random();  
        double sum = 0.0;  
        for (int i = 0; i < 1000000; i++) {  
            sum += rnd.nextDouble();  
        }  
        System.out.println("media = " + sum / 1000000.0);  
    }  
}
```



# Contenido

- Introducción histórica
- Programas y paquetes
- **Clases y objetos**
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores. Excepciones
- Cadenas de caracteres
- Arrays
- Listas
- Herencia y redefinición de comportamiento
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Clases en Java

```
class Punto {  
    private double x, y;  
    public Punto() { x = y = 0; }  
    public Punto(double a, double b) {  
        x = a; y = b;  
    }  
    public double x() { return x; }  
    public double y() { return y; }  
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }  
    public void x(double a) { x = a; }  
    public void y(double b) { y = b; }  
    public float distancia(Punto pto) {  
        return Math.sqrt(Math.pow(x - pto.x, 2)  
            + Math.pow(y - pto.y, 2));  
    }  
}
```

# Estructura de una clase en Java

```
class Partícula extends Punto {  
    final static double G = ...;  
    protected double masa;  
    public Partícula(float m) {  
        super(0, 0);  
        masa = m;  
    }  
    public Partícula(double a, double b, double m) {  
        super(a, b);  
        masa = m;  
    }  
    public void masa(double m) { masa = m; }  
    public double masa() { return masa; }  
    public double atracción(Partícula part) {  
        return G * masa * part.masa /  
            Math.pow(distancia(part), 2);  
    }  
}
```

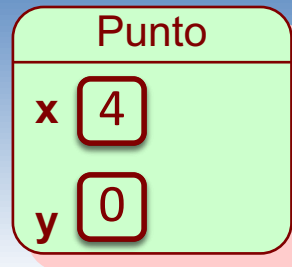
Cabecera  
Cte. de clase

Var. de instancia  
Constructor

Método de instancia

# Objetos en Java

```
public class Punto {  
    private double x, y;
```



pto

```
    public Punto(double a, double b) {  
        x = a; y = b;  
    }
```

...

```
    public void trasladar(double a, double b) {  
        x += a; y += b;  
    }
```

```
    public double distancia(Punto p) { ... }  
}
```

```
Punto pto = new Punto(1,1);
```

```
pto.trasladar(3,-1);
```

*trasladar(3,-1)*

# Variables y Métodos de instancia

- Cada instancia (objeto) tiene sus propias **variables de instancia**.
  - Ej. cada punto tiene dos variables de instancia : `x`, `y`
- Todas las instancias de una clase comparten los **métodos de instancia**
- Las **variables de instancia** (o atributos) se acceden etiquetándolas con el **nombre o la referencia de la instancia** (**this** para el objeto implicado, aunque se puede suprimir si no hay conflicto de nombres).
  - Ej. `pto.x`
  - Ej. `this.x` (o directamente `x`)
- Los **métodos de instancia** se invocan etiquetándolos con el **nombre o la referencia de la instancia** (**this** para el objeto implicado, aunque se puede suprimir si no hay conflicto de nombres).
  - Ej. `p1.distancia(p2)`
  - Ej. `this.distancia(part)` (o directamente `distancia(part)`)
- Desde fuera de una clase, habrá que tener en cuenta el control de **visibilidad** (se verá más adelante)

# Variables y métodos de instancia

```
public double distancia(Punto pto) {  
    return Math.sqrt(Math.pow(x - pto.x, 2) +  
        Math.pow(y - pto.y, 2));  
}
```

Propias del receptor.  
Puede usarse **this.x**

Otro objeto de  
la misma clase

```
Punto pto = new Punto(3,5);  
pto.distancia(new Punto(1,4));
```

```
{x = 3, y = 5, pto.x = 1, pto.y = 4}
```

# Variables y Métodos de clase

- Las **variables de clase**
  - Existen aunque no se hayan creado objetos de la clase.
  - Son comunes a todos los objetos de la clase.
  - Se declaran como **static**.
  - Se acceden etiquetando sus nombres con el nombre de la clase (aunque también se pueden etiquetar con el nombre de alguna instancia).
  - Dentro de la propia clase se acceden directamente (o etiquetando sus nombres con **this**, si es necesario)
- Los **métodos de clase**
  - Existen aunque no se hayan creado objetos de la clase.
  - Se declaran como **static**.
  - Se invocan etiquetando sus nombres con el nombre de la clase (aunque también se pueden etiquetar también con el nombre de alguna instancia).
  - Dentro de la propia clase se invocan directamente (o etiquetando sus nombres con **this**, si es necesario)

# Variables y métodos de clase

```
public double distancia(Punto pto) {  
    return Math.sqrt(Math.pow(x - pto.x, 2) +  
        Math.pow(y - pto.y, 2));  
}
```



Métodos de clase (static)  
de la clase Math



# Variables y métodos de clase





```
class Vuelo {  
  
    static private int sigVuelo = 1; // Variable de instancia  
    private String localizadorVuelo; // Variable de clase  
  
    public Vuelo(String lin) {  
        localizadorVuelo = lin + "_" + sigVuelo;  
        sigVuelo++;  
    }  
    ...  
}
```

```
Vuelo v1 = new Vuelo("Iberia"); // Iberia_1  
Vuelo v2 = new Vuelo("Lufhtansa"); // Lufhtansa_2  
Vuelo v3 = new Vuelo("Iberia"); // Iberia_3
```

# Control de la visibilidad









Existen cuatro niveles de visibilidad:








- **private** – visibilidad dentro de la propia clase
- **protected** – visibilidad dentro del paquete y de las clases herederas
- **public** – visibilidad desde cualquier paquete
- Por omisión – visibilidad dentro del propio paquete (package)

			Mismo paquete		Otro paquete	
			Subclase	Otra	Subclase	Otra
	-	<b>private</b>	NO	NO	NO	NO
	~	<b>package</b>	SÍ	SÍ	NO	NO
	#	<b>protected</b>	SÍ	SÍ	SÍ	NO
	+	<b>public</b>	SÍ	SÍ	SÍ	SÍ






# Símbolos en Eclipse

## Demo




 varInstanciaPrivada: int  
 varInstanciaProtegida: int  
 varInstanciaPaquete: int  
 varInstanciaPublica: int  
  varClasePrivada: int  
  CONS\_CLASE\_PRIVADA: int

 Demo()  
 metodoDeInstanciaPrivado():int  
 metodoDeInstanciaProtegido():int  
 metodoDeInstanciaPaquete():int  
 metodoDeInstanciaPublico():int  
  metodoDeClasePrivado():int






## DemoClaseAbstracta

 DemoClaseAbstracta()  
  metodoInstanciaAbstractoPublico():int  
  metodoClaseAbstrcotoPublico():int

## DemoInterface

  CONSTANTE: String  
 metodoInstanciaPublico():int

## DemoEnum

  Blanca: DemoEnum  
  Negra: DemoEnum  
 DemoEnum()

# Símbolos en IntelliJ

C Demo		
f	varInstanciaPrivada	int
f	varInstanciaProtegida	int
f	varInstanciaPaquete	int
f	varInstanciaPublica	int
f	varClasePrivada	int
f	CONS_CLASE_PRIVADA	int
m	Demo()	
m	metodoDeClasePrivado()	int
m	metodoDeInstanciaPaquete()	int
m	metodoDeInstanciaPrivado()	int
m	metodoDeInstanciaProtegido()	int
m	metodoDeInstanciaPublico()	int

C DemoClaseAbstracta		
m	DemoClaseAbstracta()	
m	metodoClaseAbstrctoPublico()	int
m	metodoInstanciaAbstrctoPublico()	int

E DemoEnum		
f	Blanca	
f	Negra	
m	DemoEnum()	
m	valueOf(String) DemoEnum	
m	values() DemoEnum[]	

I DemoInterface		
f	CONSTANTE	String
m	metodoInstanciaPublico()	int

# La vida de los objetos

- Los objetos son siempre instancias de alguna clase.
- Durante la ejecución de un programa
  - Se crean objetos
  - Interactúan entre sí (enviándose mensajes)
  - Se eliminan los objetos no necesarios
    - La eliminación es automática (recolección de basura automática)

# Creación de objetos

- Los objetos hay que crearlos
  - Se les reserva espacio en memoria
  - Se asignan unos valores iniciales a sus variables de estado
  - Se debe utilizar un constructor:  
**new <constructor>(<lista args>)**
- El operador **new** devuelve una referencia al objeto que crea.
  - Puede asignarse a una variable  
**pto = new Punto(3, 4) ;**
  - Puede usarse en una expresión  
**pto.distancia(new Punto(2,3)) ;**

# Constructores de objetos

- Una clase puede definir varios constructores
  - Con distinto número de argumentos y/o
  - Con argumentos de distintos tipos
- Si no se ha definido ningún constructor, entonces existe un constructor “por defecto”
  - Si hay constructores, el constructor “por defecto” no se crea.

# Variables de objeto

- Las variables se declaran de una clase (o interfaz)

```
Punto pto;
```

Declaración

pto

- Es una referencia a un objeto, **NO** es un objeto
  - No puede recibir aún mensajes.**
- Una variable puede referenciar a un objeto (Asignación)

```
pto = new Punto(3, 4);
```

Asignación

- Ya puede recibir mensajes.
- Estos dos pasos se pueden realizar simultáneamente:

```
Punto pto = new Punto(3, 4);
```

inicialización



# Uso de objetos. Acceso al estado

- A las variables de estado (atributos) de un objeto se puede acceder mediante expresiones como:

**pto.x**

- Para ello, el atributo **x** debe estar definido en la clase a la que pertenezca **pto**, con la visibilidad adecuada.
- Dentro de la misma clase donde se define el atributo, se puede utilizar la pseudovariante **this**

```
public Punto(double x, double y) {  
    this.x = x;  
    this.y = y;  
}
```

- Se suele utilizar para evitar conflictos. Si no hay conflicto de nombres, **this** suele suprimirse.

```
public Punto(double abs, double ord) {  
    x = abs;  
    y = ord;  
}
```

- Se supone que un objeto debe proteger su estado:
  - El acceso directo a las variables de estado de un objeto por parte de otro de otra clase no es recomendable

# Uso de objetos. Invocación de métodos

- Invocación de los métodos
  - Los métodos cuando son visibles, se invocan mediante expresiones como:  
`pto.trasladar(2, 2);`
  - Para ello, **pto** debe hacer referencia a un objeto.
  - El compilador admitirá la expresión si el **tipo estático** del receptor “sabe responder” a ese mensaje.
  - El código concreto del método invocado en un mensaje dependerá del **tipo dinámico** del objeto receptor.

# Asignación o Copia

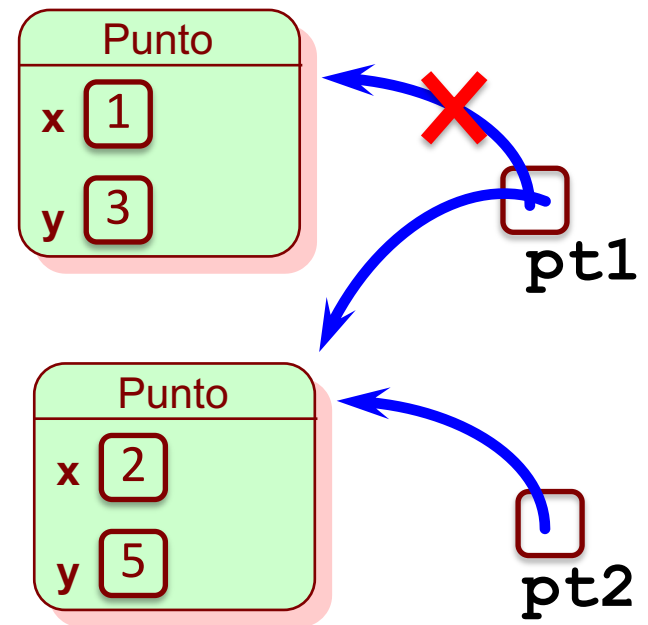
- Una variable que referencia a un objeto se puede **asignar** a otra de su misma clase. En tal caso se copia la referencia y ambas compartirán el mismo objeto.
- Para duplicar un objeto se debe crear otro de la misma clase y **copiar** sus variables de estado.
- También se puede incluir un constructor (o un método) de copia en la clase correspondiente.
- O utilizar el método **clone ()** de la clase **Object** de **java.lang**

# Asignación o Copia

- La asignación entre referencias no realiza copias de objetos, sino que hace que éstos se “compartan”:

```
Punto pt1 = new Punto(1,3);  
Punto pt2 = new Punto(2,5);
```

```
pt1 = pt2;
```

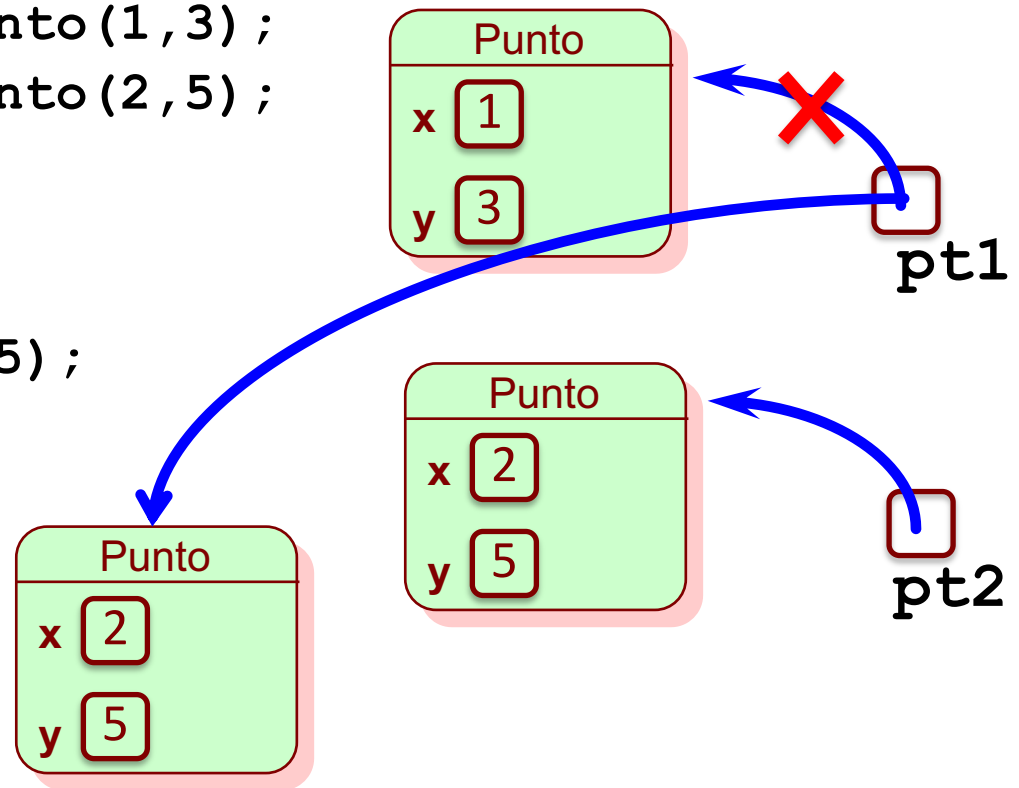


# Asignación o Copia

- Para que se produzca una copia, hay que crearla explícitamente:

```
Punto pt1 = new Punto(1,3);  
Punto pt2 = new Punto(2,5);
```

```
pt1 = new Punto(2,5);
```



# Uso de objetos (asignación)

```
class Segmento {  
    private Punto origen, extremo;  
    public Segmento(Punto pto1, Punto pto2) {  
        origen = pto1;  
        extremo = pto2;  
    }  
    ... // Otros métodos  
    public float longitud() {  
        return origen.distancia(extremo);  
    }  
}
```

# Uso de objetos (copia)

```
class Segmento {  
    private Punto origen, extremo;  
    public Segmento(Punto pto1, Punto pto2) {  
        origen = new Punto(pto1.x(), pto1.y());  
        extremo = new Punto(pto2.x(), pto2.y());  
    }  
    ... // Otros métodos  
    public float longitud() {  
        return origen.distancia(extremo);  
    }  
}
```

# Eliminación de objetos

- La eliminación de objetos es automática en Java, y se produce cuando el objeto es inalcanzable; es decir, no existen referencias a él.



# Contenido

- Introducción histórica
- Programas y paquetes
- Clases y objetos
- **Elementos del lenguaje:**
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores. Excepciones
- Cadenas de caracteres
- Arrays
- Listas
- Herencia y redefinición de comportamiento
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Tipos y valores

- En Java se distingue entre tipos básicos y tipos “clase”.
  - Llamamos **valores** a los datos de estos tipos básicos.
  - Llamamos **objetos** a las instancias de los tipos clase.
- Sólo existen los siguientes tipos básicos:

**byte** (entero de 8 bits)

**int** (entero de 32 bits)

**float** (decimal de 32 bits)

**char** (Unicode de 16 bits)

**short** (entero de 16 bits)

**long** (entero de 64 bits)

**double** (decimal de 64 bits)

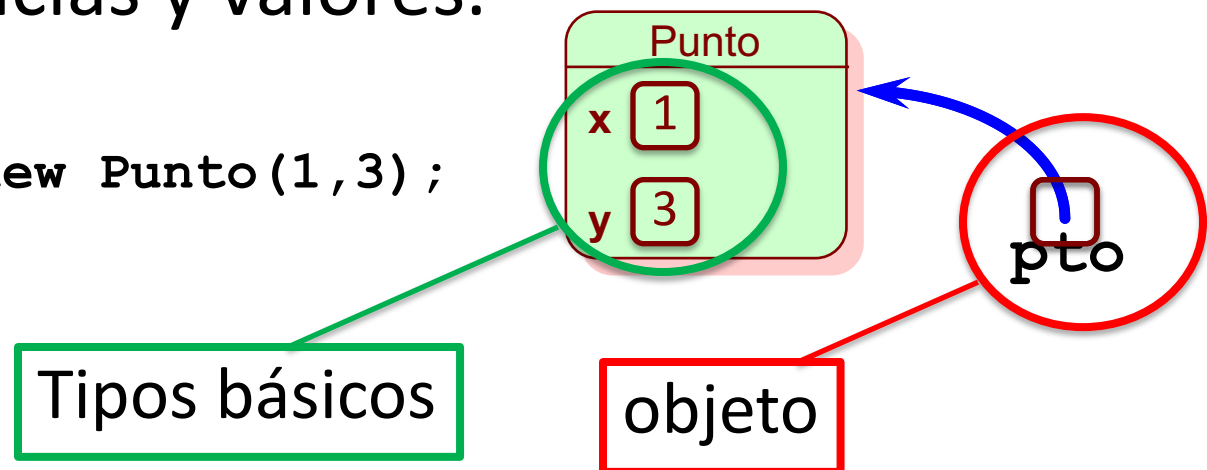
**boolean** (**true**, **false**)

- No se pueden definir más tipos básicos.

# Tipos básicos frente a clases

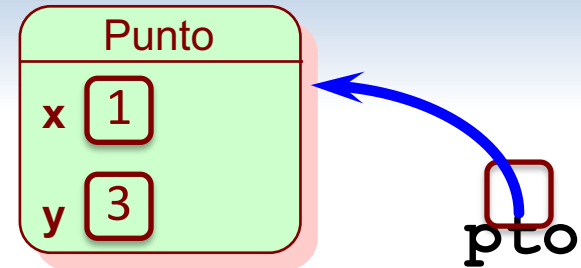
- Variables de tipos básicos
  - Almacenan el **valor**
- Variables de objetos
  - Almacenan la **referencia al objeto**
- Esto tiene consecuencias en la manipulación de referencias y valores.

```
Punto pto = new Punto(1,3);
```

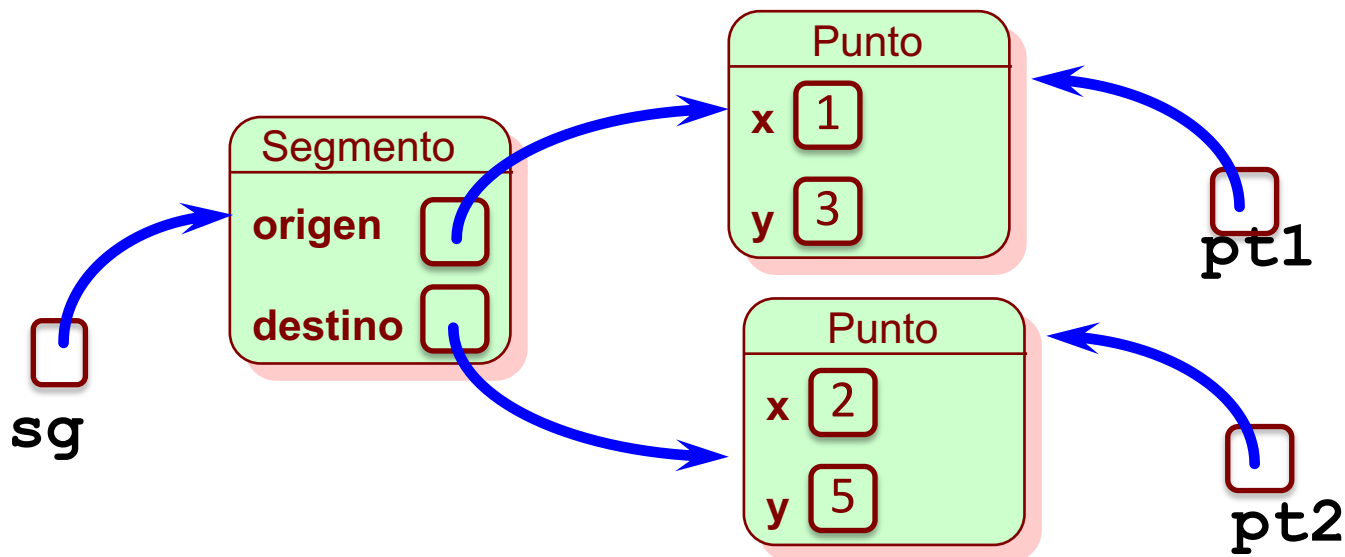


# Tipos básicos frente a clases

```
Punto pto = new Punto(1,3);
```



```
Punto pt1 = new Punto(1,3);  
Punto pt2 = new Punto(2,5);  
Segmento sg = new Segmento(pt1,pt2);
```



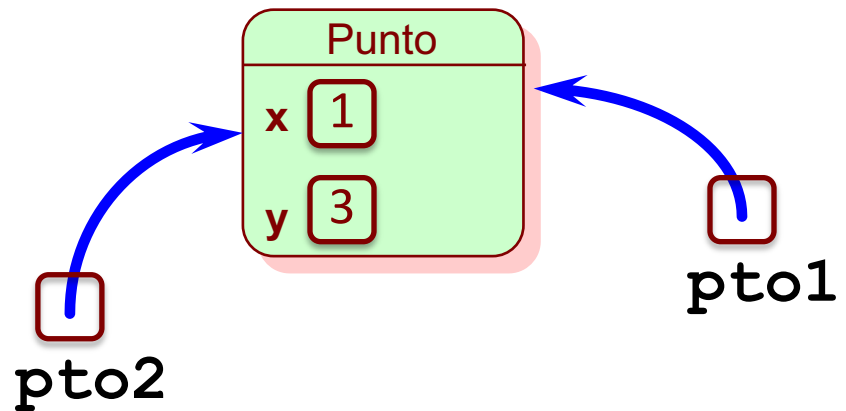
# Tipos básicos frente a Clases

## asignación

```
int a, b;  
b = 3;  
a = b;
```



```
Punto pto1, pto2;  
pto1 = new Punto(1,3);  
pto2 = pto1;
```



# Tipos básicos frente a Clases

## comparación

```
int a, b;  
b = 3;  
a = b;
```

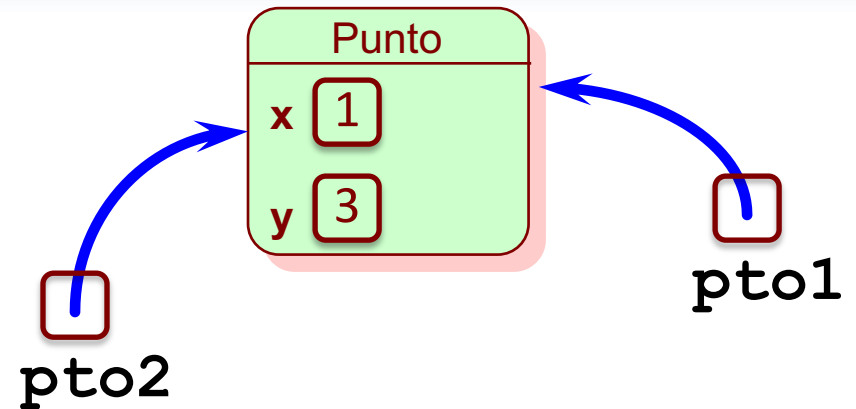
b 3  
a 3

```
if (a == b) { // true  
    ...  
}
```

# Tipos básicos frente a Clases

## comparación

```
Punto pto1, pto2;  
pto1 = new Punto(1,3);  
pto2 = pto1;
```



```
if (pto1 == pto2) { // true  
    ...  
}
```

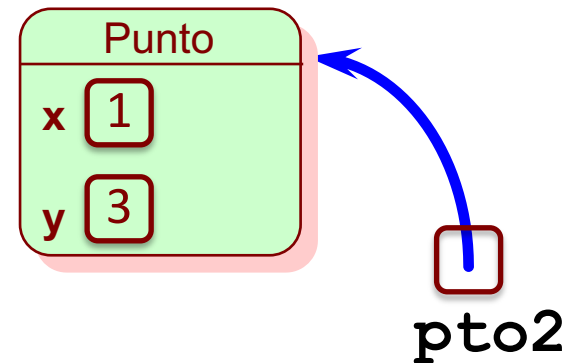
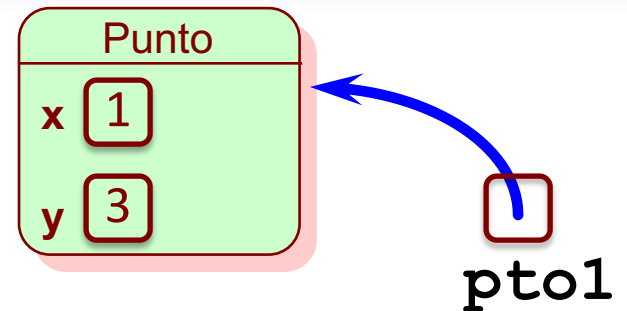
# Tipos básicos frente a Clases

## comparación

```
Punto pto1, pto2;  
pto1 = new Punto(1,3);  
pto2 = new Punto(1,3);
```

```
if (pto1 == pto2) { // false  
    ...  
}
```

Compara referencias





# Tipos básicos frente a Clases

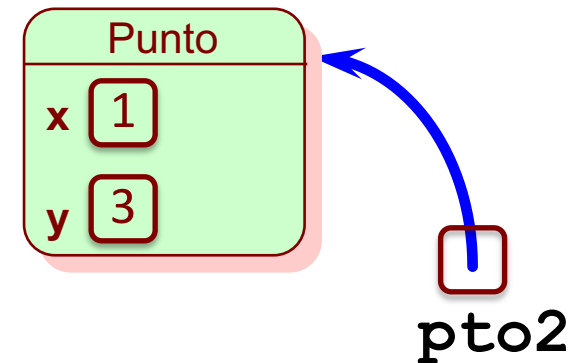
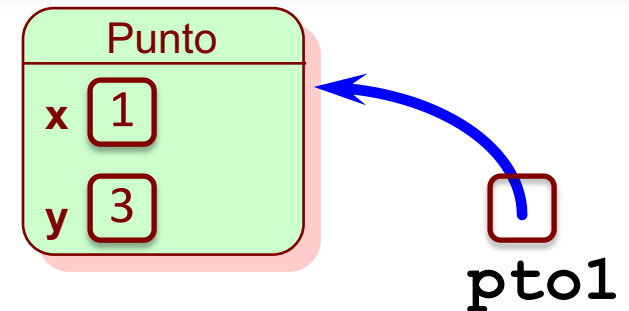
## comparación

```
Punto pto1, pto2;  
pto1 = new Punto(1,3);  
pto2 = new Punto(1,3);
```

Aunque sería mejor usar  
`equals` (tema 4)

```
if (pto1.abscisa() == pto2.abscisa()  
    && (pto1.ordenada() == pto2.ordenada())) { // true  
    ...  
}
```

Compara contenido



# Tipos básicos frente a Clases

## comparación

- Norma general:
  - Para **comparar tipos básicos** usaremos `==`
  - Para **comparar objetos** siempre usaremos el método `equals`.

```
Punto pto1, pto2;  
pto1 = new Punto(1,3);  
pto2 = new Punto(1,3);
```

```
if (pto1.abscisa() == pto2.abscisa()  
    && (pto1.ordenada() == pto2.ordenada())) { // true  
    ...  
}
```

```
if (pto1.equals(pto2) { // de momento es equivalente a pto1 == pto2  
    ...                // más adelante redefiniremos adecuadamente este método  
}
```

# Conversiones de tipos y clases

- Se producen conversiones de tipo o de clase de forma **implícita** en ciertos contextos.

- Siempre a **tipos más “amplios”** siguiendo la ordenación:

- `byte ->      short -> int -> long -> float -> double`  
`char ->`

- o a **clases ascendentes** en la línea de la herencia.

- Se permiten conversiones **explícitas** en sentido contrario mediante la construcción:

- `(<tipo/clase>) <expresión>`

- Las conversiones explícitas se denominan “**casting**” y si no se realizan adecuadamente pueden provocar errores en ejecución, por lo que es importante evitarlas.

# Conversiones implícitas: contextos

- La conversión implícita se produce en los siguientes contextos:
  - **Asignaciones** (el tipo de la expresión se *promociona* al tipo de la variable de destino)
  - **Invocaciones de métodos** (los tipos de los parámetros reales se promocionan a los tipos de los parámetros formales)
  - **Evaluación de expresiones aritméticas** (los tipos de los operandos se promocionan al del operando con el tipo más general y, como mínimo se promocionan a `int`)
  - **Concatenación de cadenas** (los valores de los argumentos se convierten en cadenas)

# Variables

- Las variables se utilizan tanto para referirse a objetos como a valores.
- Antes de usar una variable se requiere una declaración:

`<tipo> <identificador>`

**`int contador;`**

- En ambos casos las variables se pueden inicializar mediante una sentencia de asignación.
- Declaración e inicialización pueden aparecer en la misma línea

**`int contador = 0;`**

# Constantes

- Una variable se puede declarar como constante precediendo su declaración con la etiqueta **final**:  
**final int MAXIMO = 0;**
- La inicialización de una constante se puede hacer en cualquier momento posterior a su declaración (salvo si son constantes de clase (**static**)).

**final int MAXIMO;**

...

**MAXIMO = 0;**

- Cualquier intento de cambiar el valor de una variable **final** después de su inicialización produce un error en tiempo de compilación.

# Identificadores

- Un *identificador* (nombre) es una secuencia arbitraria de caracteres Unicode: letras, dígitos, subrayado o símbolos de monedas. No debe comenzar por dígito ni coincidir con alguna palabra reservada.

```
int número;
```

- Los identificadores dan nombre a variables, métodos, clases e interfaces.
- Por convenio:
  - Nombres de variables y métodos en minúsculas. Si son compuestos, las palabras posteriores no se separan y comienzan con mayúscula.

```
long valorMáximo;
```

- Nombres de clase igual, pero comenzando con mayúscula.

```
class PuntoAcotado ...
```

- Nombres de constantes todo en mayúsculas. Si son compuestos, las palabras se separan con subrayados.

```
final int CTE_GRAVITACIÓN;
```

# Ámbito de una variable

- Un identificador debe ser único dentro de su ámbito.
- El **ámbito** de una variable es la zona de código donde se puede usar su identificador sin cualificar.
- El ámbito determina cuándo se crea y cuándo se destruye espacio de memoria para la variable.
- Las variables, según su ámbito, se clasifican en las siguientes categorías:
  - Variable de clase o de instancia
  - Variable local
  - Parámetro de método
  - Parámetro de gestor de excepciones



# Ámbitos

class MiClase { ...

Variables de clase

Parámetros y

~~Variables de instancia~~

Variables locales

Parámetros

catch

static public void método(...) {

{...}

}

catch(...) {

}

}

}

# Inicialización de variables

- Las **variables de clase** se inicializan automáticamente al *cargar* la clase.
- Las **variables de instancia** se inicializan automáticamente **cada vez que se crea una instancia**.
- Las **variables locales** no se inicializan de forma automática y el compilador produce un error si no se hace explícitamente.
- Valores de inicialización automática:

`false '\u0000' 0 +0.0F +0.0D null`

# Expresiones

- Una **expresión** es una combinación de
  - literales,
  - variables,
  - operadores y
  - mensajes,

acorde con la sintaxis del lenguaje, que se

- *evalúa* a un valor simple o
- a una referencia a un objeto y

**devuelve el resultado** obtenido.

# Operadores (I)

- Un **operador** es una función de uno, dos o tres argumentos.
- Existen operadores
  - aritméticos  $(+_ , -_ , _++ , _-- , ++_ , --_)$   
 $(+_ , -_ , *_ , _/_ , _\%_)$
  - de relación/comparación  $(> , >= , < , <= , == , !=)$
  - lógicos  $(\&\& , || , ! , \& , | , ^)$
  - de asignación  $(= , += , -= , *= , /= , \% = , \&= , |= , ^=)$
  - para la manipulación de bits
  - Otros operadores  $(\_? \_ : \_)$

# Operadores (II)

- Con un operador y sus argumentos se construyen expresiones simples.

`3 * 5      x += 7.3    'a' <= 45`



- Las expresiones simples se pueden combinar dando lugar a expresiones compuestas.

`x += 7.3 + 3 * 5`

- El orden de evaluación de las expresiones compuestas depende de:
  - La precedencia y
  - La asociatividad de los operadores que aparezcan

# Precedencia de operadores

- **Precedencia** (en sentido decreciente)

	<code>var++</code>	<code>var--</code>			<code>&amp;</code>	
	<code>++var</code>	<code>--var</code>	<code>!</code>		<code> </code>	
	<code>new</code>	<code>(tipo)</code>	<code>exp</code>		<code>&amp;&amp;</code>	
	<code>*</code>	<code>/</code>	<code>%</code>		<code>  </code>	
	<code>+</code>	<code>-</code>			<code>=</code>	<code>+=</code> <code>-=</code> <code>*=</code> <code>...</code>
	<code>&lt;</code>	<code>&gt;</code>	<code>&lt;=</code> <code>&gt;=</code>			
	<code>==</code>	<code>!=</code>				

- El orden de las operaciones en una expresión siempre se puede modificar mediante el uso de paréntesis

# Asociatividad de operadores

- **Asociatividad**

- Todos los operadores binarios (excepto la asignación) a igualdad de precedencia, asocian por la izquierda.
- La asignación asocia por la derecha

- Ejemplos de expresiones:

`3 + 4 / 2`      `3 * (x = 5)`

`x = y = 3`      `x = ++y / 2`

`x > 3 && y`      `x = y++ / 2`

# Instrucciones/sentencias

- Existen tres clases de instrucciones o sentencias:
  - Sentencias de expresión – Se obtienen terminando en ‘;’ alguna de las expresiones siguientes:
    - asignaciones
    - incrementos/decrementos ++/--
    - mensajes
    - creaciones de objeto
  - Sentencias de declaración de variables
  - Sentencias de control



# Instrucciones de declaración

- Las instrucciones de declaración de variables tienen la forma: <tipo> <n. variable>

```
int x;
```

- Las declaraciones de variables del mismo tipo/clase pueden agruparse:

```
int x, y, z;
```

- Las sentencias de declaración pueden agruparse con las de asignación a las mismas variables:

```
int x = 5, y = 12, z = 213;
```

# Sentencias de control

- Las sentencias de control del flujo de ejecución se agrupan en:
  - sentencias de repetición
  - sentencias de selección
  - sentencias para el control de excepciones
  - sentencias de salto/ramificación

# Sentencias de repetición

```
while (<exp. booleana>
    <sentencia>
```

```
do
```

```
    <sentencia>
```

```
while (<exp. booleana>);
```

```
for (<exp1>; <exp. bool>; <exp2>)
    <sentencia>
```

Existe una sintaxis de **for** especial para arrays y colecciones.

# Sentencias de selección (I)

```
if (<exp. bool>) <sentencia>
```

```
if (<exp. bool>) <sentencia1>  
else <sentencia2>
```

```
if (<exp. bool1>) <sentencia1>  
else if (<exp. bool2>) <sentencia2>
```

```
..  
else <sentenciaN>
```

```
<exp bool> ? <exp1> : <exp2>
```

Ejemplo:

```
System.out.println("El carácter " + car + " es " +  
    (Character.toUpperCase(car) ? "mayúscula" : "minúscula"));
```

# Sentencias de selección (II)

```
switch (<exp>) {  
    case <altern1>: <sent1>; break;  
    case <altern2>: <sent2>; break;  
    ...  
    case <alternk>: <sentk>; break;  
    default: <sentD>; break;  
}
```

- <exp> debe ser de tipo char, byte, short, int o String

# Sentencias de selección (III)

```
int año, mes, numDías;  
...  
switch (mes) {  
    case 1:  
    case 3:  
    case 5:  
    case 7:  
    case 8:  
    case 10:  
    case 12: numDías = 31; break;  
    case 4:  
    case 6:  
    case 9:  
    case 11: numDías = 30; break;  
    case 2: if ... else ...  
}
```

# Contenido

- Introducción histórica
- Programas y paquetes
- Clases y objetos
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- **Control de errores. Excepciones**
- Cadenas de caracteres
- Arrays
- Listas
- Herencia y redefinición de comportamiento
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Control de excepciones (I)

- Mecanismo de ayuda para la comunicación y el manejo de errores
- Cuando se produce un error en un método:
  1. Se crea un objeto (el sistema o el programador con `new`) de la clase **Exception**, **RuntimeException** (de `java.lang`) o de alguna clase heredera de ellas, con información sobre el error (normalmente una cadena de caracteres como parámetro)
  2. Se lanza (el sistema o el programador mediante la instrucción `throw`), dicha excepción (objeto)

Por ejemplo:

```
throw new RuntimeException("Error...") ;
```

3. Se interrumpe el flujo normal de ejecución
4. El entorno de ejecución intenta encontrar un tratamiento (captura) para dicho objeto.
  - a. dentro del propio método o
  - b. en alguno de los anteriores en las sucesivas invocaciones
5. Si no se encuentra un tratamiento, el programa finaliza con error



# Control de excepciones (II)

Existen tres sentencias relacionadas con el control de excepciones:

- **try**

delimita un bloque de instrucciones donde se puede producir una excepción,

- **catch**

identifica un bloque de código asociado a un bloque **try** donde se trata un tipo particular de excepción,

- **finally**

identifica un bloque de código que se ejecutará después de un bloque **try** con independencia de que se produzcan o no excepciones.

# Control de excepciones (III)

El aspecto normal de un segmento de código con control de excepciones sería el siguiente:

```
try {  
  <sentencia/s>  
} catch (<tipoexcepción> <identif>) {  
  <sentencia/s>  
}  
...  
} catch (<tipoexcepción> <identif>) {  
  <sentencia/s>  
} finally {  
  <sentencia/s>  
}
```

The diagram illustrates the components of a try-catch-finally block with three callouts:

- A red callout labeled "Bloque vigilado" (Monitored block) points to the code inside the **try** block.
- A green callout labeled "Manejador" (Handler) points to the code inside the first **catch** block.
- A green callout labeled "Manejador" (Handler) points to the code inside the second **catch** block.
- A blue callout labeled "Siempre se ejecuta" (Always executes) points to the code inside the **finally** block.

# Contenido

- Introducción histórica
- Programas y paquetes
- Clases y objetos
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores. Excepciones
- **Cadenas de caracteres**
- Arrays
- Listas
- Herencia y redefinición de comportamiento
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Cadenas de caracteres. Clase String

- Secuencia de caracteres Unicode
  - Entre comillas dobles:  
"Ejemplo de cadena de caracteres"
- La clase `String` dispone de constructores y métodos para crear y manipular cadenas de caracteres:

```
String cadena = "hola";  
cadena = "adios";  
String saludo = new String("hola y adios");
```

# Cadenas de caracteres

- Los métodos más básicos de String son:
  - length() – devuelve el número de caracteres de la cadena
  - charAt(i) – devuelve el carácter de la posición i en la cadena (el primer carácter ocupa la posición 0)
- Los objetos de la clase **String** en Java son **objetos inmutables**
- Si se intenta acceder a una posición no válida el sistema lanza una excepción:
  - StringIndexOutOfBoundsException

# Cadenas de caracteres

- En la concatenación pueden intervenir otros tipos de datos

```
int i = 42;  
System.out.println("i es " + i);
```

SALIDA:            i es 42

- Si no son tipos básicos ...

```
Punto p = new Punto(3,4);  
  
System.out.println("p es " + p);
```

SALIDA:                    p es Punto@119c0982

# Cadenas de caracteres

- Para evitar esto hay que incluir en las clases el método `toString()`

```
class Punto {  
    ...  
    public String toString() {  
        return "(" + x + "," + y + ")";  
    }  
}
```

- Ahora

```
Punto p = new Punto(3,4);
```

```
System.out.println("p es " + p);
```

SALIDA:                      p es (3,4)

# Cadenas de caracteres

- Comparación de cadenas:
  - `c1.equals(c2)` – devuelve `true` si `c1` y `c2` son iguales y `false` en otro caso.
  - `c1.equalsIgnoreCase(c2)` – Igual que la anterior, pero sin tener en cuenta las diferencias por mayúsculas y minúsculas.
  - `c1.compareTo(c2)` – devuelve un entero menor, igual o mayor que cero cuando `c1` es menor, igual o mayor que `c2`.
  - `c1.compareToIgnoreCase(c2)` – Igual que la anterior, pero sin tener en cuenta las diferencias por mayúsculas y minúsculas.
- ¡ojo!
  - `c1 == c2`, `c1 != c2`, ... (operadores relacionales) comparan variables referencia. Recordad que la igualdad de objetos siempre se hacen con `equals`.



# Contenido

- Introducción histórica
- Programas y paquetes
- Clases y objetos
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores. Excepciones
- Cadenas de caracteres
- **Arrays**
- Listas
- Herencia y redefinición de comportamiento
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Arrays en Java (I)

componentes

array

nombre

- Objetos que representan estructuras de datos de longitud fija con componentes de un mismo tipo o clase.
- Con una sintaxis particular:

➤ Declaración

```
int [] listaEnteros;  
Punto [] listaPuntos;
```

➤ Inicialización

```
listaEnteros = new int[10];  
listaPuntos = new Punto[23];  
char[] vocales = {'a', 'e', 'i', 'o', 'u'};  
Punto p = new Punto(1, 1);  
Punto[] ap = {new Punto(2, 2),
```

componentes

tamaño

array literal

- La longitud se guarda en la constante **length**.

# Arrays

- Los arrays siempre comienzan por la posición 0

```
for (int i = 0; i < listaEnteros.length; i++) {  
    listaEnteros[i] = i;  
}
```

```
for (int i = 0; i < listaPuntos.length; i++) {  
    listaPuntos[i] = new Punto(i, i);  
}
```

```
String[] cadenas = {"CAD1", "CAD2", "CAD3"};  
for (int i = 0; i < cadenas.length; i++) {  
    System.out.println(cadenas[i].toLowerCase());  
}
```

- Si se intenta acceder a una posición no válida el sistema lanza una excepción:
  - `ArrayIndexOutOfBoundsException`

# Recorridos de arrays

```
char[] arrayOrigen =  
    { 'd', 'e', 's', 'c', 'a', 'f', 'e', 'i', 'n', 'a', 'd', 'o' };  
  
for (int i = 0; i < arrayOrigen.length; i++) {  
    System.out.println(arrayOrigen[i]);  
}
```

- Es posible utilizar una sintaxis alternativa

```
for( char c : arrayOrigen) {  
    System.out.println(c);  
}
```

- Existe una clase `Arrays` en el paquete `java.util` que proporciona métodos de utilidad para manipular arrays

# Copia de arrays (I)

- Para la copia eficiente de componentes de un array a otro, Java tiene el método **arraycopy** en la clase **System**.

```
public static void arraycopy( Object arrayOrigen,  
                             int  primÍndiceOrigen,  
                             Object arrayDestino,  
                             int  primÍndiceDestino,  
                             int  númeroDeCompCopia)
```

```
char[] arrayOrigen =  
{'d','e','s','c','a','f','e','i','n','a','d','o'};  
char[] arrayDestino = new char[7];
```

```
System.arraycopy(arrayOrigen, 3, arrayDestino, 0, 7);  
System.out.println(new String(arrayDestino));
```

# Arrays incompletos

- Hay situaciones en las que los arrays no son estructuras adecuadas:
  - Cuando no sabemos cuántos datos vamos a almacenar.
    - Pues hay que hacer una reserva por defecto y llevar un contador de los datos que hay.
  - Pueden aparecer mas datos que hace que se quede pequeño el array reservado.
    - Se necesita hacer crecer al array.

Para eso existen otras soluciones: Listas

# Contenido

- Introducción histórica
- Programas y paquetes
- Clases y objetos
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores. Excepciones
- Cadenas de caracteres
- Arrays
- **Listas**
- Herencia y redefinición de comportamiento
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Listas

- Alternativa al array mucho mas fácil de manipular:
  - Array
    - Hay que crearlo de un tamaño dado. Vigilar siempre si hay espacio para añadir un elemento mas.
    - Hay que llevar un contador para saber cuántos elementos tenemos.
    - Si queremos insertar o borrar un elemento hay que mover todos desde esa posición hasta el final para no dejar huecos.
  - Las listas hacen todo eso por nosotros.



# Listas

- Para declarar una variable que referencia a una lista hacemos (Se tratará a fondo en el tema 6)

```
ArrayList<T> lista;
```

Entre <> indicamos qué clase de datos guardar (No pueden ser tipos básicos). Una lista siempre contiene objetos:

Para guardar enteros el tipo debe ser Integer. Para guardar doubles el tipo debe ser Double: (lo veremos en el tema 4)

```
ArrayList<String> ls = new ArrayList<>();  
ArrayList<Integer> li = new ArrayList<>();
```

Regla del diamante:

Si ya hemos indicado el tipo en la declaración de la variable no hace falta volver a declararlo en la construcción del objeto.

# Listas

- Hay que importar la clase para poder usarla:

```
import java.util.ArrayList;
```

- Crear una lista y añadirle elementos:

```
ArrayList<String> lista = new ArrayList<>();
```

```
lista.add("hola");
```

```
lista.add("que");
```

```
lista.add("tal");
```

```
lista.add(1, "amigo");
```

```
System.out.println(lista.get(2)); // que
```

```
System.out.println(lista.size()); // 4
```

```
lista.remove(0);
```

```
lista.set(1, "mio");
```

```
System.out.println(lista);
```

```
// ["amigo", "mio", "tal"]
```

hola							
hola		que					
hola		que		tal			
hola		amigo		que		tal	
; // que							
; // 4							
amigo		que		tal			
amigo		mio		tal			

# Listas. Recorrido

- Dos formas de recorrerla completamente:

```
ArrayList<String> lista = new ArrayList<>();  
lista.add("hola");  
lista.add("que");  
lista.add("tal");
```

```
for (int i = 0; i < lista.size(); i++) {  
    System.out.println(lista.get(i));  
}
```

```
// Mejor con un foreach  
for(String s : lista) {  
    System.out.println(s);  
}
```

# Clases anidadas

- Se definen dentro del cuerpo de otra clase.
- Se pueden distinguir diversos tipos de clases anidadas (internas, locales, anónimas), pero solo consideraremos:
  - clases internas estáticas
- Una clase interna estática es la que se define como un atributo más de la clase, y en la que se utiliza el calificador `static`.
- Para acceder a ellas (si su visibilidad lo permite) debe cualificarse con el nombre de la clase externa.

# Clases internas estáticas

- Un ejemplo de clase interna estática son los datos enumerados.

```
public class Urna {  
  
    static public enum ColorBola {Blanca, Negra};  
  
    private int numBlancas, numNegras;  
  
    public Urna(int nB, int nN {  
        numBlancas = nB;  
        numNegras  = nN;  
    }  
  
    public ColorBola sacaBola(){  
        ColorBola bolaSacada = null;  
        if (...) {  
            bolaSacada = ColorBola.Blanca;  
            numBlancas--;  
        } else {  
            bolaSacada = ColorBola.Negra;  
            numNegras--;  
        }  
        return bolaSacada;  
    }  
    ...  
}
```

```
...  
Urna.ColorBola cb = Urna.ColorBola.Negra;  
...
```

# Contenido

- Introducción histórica
- Programas y paquetes
- Clases y objetos
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores. Excepciones
- Cadenas de caracteres
- Arrays
- Listas
- **Herencia y redefinición de comportamiento**
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Subclasses/Herencia

- En Java se pueden definir **subclases** o clases que **heredan** estado y comportamiento de otra clase (la **superclase**) a la que amplían:

```
class MiClase extends Superclase {  
    . . .  
}
```

- En Java sólo se permite **herencia simple**.
- Todas las jerarquías confluyen en la clase **Object** de **java.lang** que recoge los comportamientos básicos que debe presentar cualquier clase.

# Herencia y constructores

- Los constructores **no** se heredan.
- Cuando se define un constructor se debe proceder de alguna de las tres formas siguientes:

- Invocar a un constructor de la misma clase (con distintos argumentos) mediante this:

- Por ejemplo:

```
public Punto() {  
    this(0,0);  
}
```

- La llamada a this debe estar en la primera línea

- Invocar algún constructor de la superclase mediante super:

- Por ejemplo:

```
public Partícula(double a, double b, double m) {  
    super(a,b);  
    masa = m;  
}
```

- La llamada a super debe estar en la primera línea.

- De no ser así, se invoca por defecto al constructor sin argumentos de la superclase:

- Por ejemplo:

```
public Partícula() {  
    // Se invoca el constructor por defecto Punto()  
    masa = 0;  
}
```



# Herencia, variables y métodos

- Métodos de instancia:
  - Un método de instancia de una clase puede redefinirse en una subclase.
    - Salvo si el método (o la clase) está declarado como **final**.
  - La resolución de los métodos de instancia se realiza por vinculación dinámica.
  - Una redefinición puede ampliar la visibilidad de un método.
  - El método redefinido queda oculto en la subclase por el nuevo método.
    - Si se desea acceder a la versión redefinida, se debe utilizar la sintaxis `super.<nombre del método>(argumentos)`
    - La resolución de una llamada a `super` se hace por **vinculación dinámica** comenzando desde la clase en la que aparece `super`.
- Métodos de clase y variables de instancia o de clase:
  - Se resuelven por vinculación estática.

# Herencia, y resolución del método a ejecutar

Dos fases:

- Compilación: Atiende al tipo estático.
  - El tipo estático tiene que ser capaz de responder al mensaje con un método suyo o de sus clases superiores.
  - Si no es así se produce un error de compilación
- Ejecución: Atiende al tipo dinámico
  - El método a ejecutar comienza a buscarse en la clase del tipo dinámico y sigue buscando de forma ascendente por las clases superiores.
  - Si ha compilado, seguro que hay un método

# Compilación y Vinculación dinámica

```
Punto pto = new Particula(3, 5, 22);
```

Tipo estático  
de **pto**

Tipo dinámico  
de **pto**

```
pto.trasladar(4,6);
```

- **Compila** porque el tipo estático **sabe** responder a ese mensaje.
- Al ejecutar **se busca** en el **tipo dinámico**. Si no se encuentra, se sube por la herencia hasta encontrarlo.
  - **Es seguro** que se encuentra porque ha compilado

```
pto.atraccion(new Particula(3,4,6));
```

- **No compila** porque el tipo estático **no sabe** responder a ese mensaje.

# Contenido

- Introducción histórica
- Programas y paquetes
- Clases y objetos
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores. Excepciones
- Cadenas de caracteres
- Arrays
- Listas
- Herencia y redefinición de comportamiento
- **Polimorfismo y vinculación dinámica**
- Clases abstractas e interfaces

```

class Uno {
    public int test() { return 1; }
    public int resultado1() {
        return this.test();
    }
}

```

```

class Dos extends Uno {
    public int test() { return 2; }
}

```

```

class Tres extends Dos {
    public int resultado2() { return this.resultado1(); }
    public int resultado3() { return super.test(); }
}

```

```

class Cuatro extends Tres {
    public int test() { return 4; }
}

```

```

class Prueba {
    public static void main(String [] args) {
        Tres obj3 = new Tres();
        Cuatro obj4 = new Cuatro();
        System.out.println("obj3.test()          = " + obj3.test());
        System.out.println("obj3.resultado2() = " + obj3.resultado2());
        System.out.println("obj3.resultado3() = " + obj3.resultado3());
        System.out.println("obj4.resultado1() = " + obj4.resultado1());
        System.out.println("obj4.resultado2() = " + obj4.resultado2());
        System.out.println("obj4.resultado3() = " + obj4.resultado3());
    }
}

```

SALIDA:

```

obj3.test()          = 2
obj3.resultado2() = 2
obj3.resultado3() = 2
obj4.resultado1() = 4
obj4.resultado2() = 4
obj4.resultado3() = 2

```

```

class Uno {
    public int test() { return 1; }
    public int resultado1() {
        return this.test();
    }
}

class Dos extends Uno {
    public int test() { return 2; }
}

class Tres extends Dos {
    public int resultado2() { return this.resultado1(); }
    public int resultado3() { return super.test(); }
    public int test() { return 3; }
}

class Cuatro extends Tres {
    public int test() { return 4; }
}

class Prueba {
    public static void main(String [] args) {
        Tres obj3 = new Tres();
        Cuatro obj4 = new Cuatro();
        System.out.println("obj3.test()          = " + obj3.test());
        System.out.println("obj3.resultado2() = " + obj3.resultado2());
        System.out.println("obj3.resultado3() = " + obj3.resultado3());
        System.out.println("obj4.resultado1() = " + obj4.resultado1());
        System.out.println("obj4.resultado2() = " + obj4.resultado2());
        System.out.println("obj4.resultado3() = " + obj4.resultado3());
    }
}

```

SALIDA:

```

obj3.test()          = 3
obj3.resultado2() = 3
obj3.resultado3() = 2
obj4.resultado1() = 4
obj4.resultado2() = 4
obj4.resultado3() = 2

```

```

class Uno {
    public int test() { return 1; }
    public int resultado1() {
        return this.test();
    }
}

class Dos extends Uno {
    public int test() { return 2; }
}

class Tres extends Dos {
    public int resultado2() { return this.resultado1(); }
    public int resultado3() { return super.test(); }
    public int test() { return 3; }
}

class Cuatro extends Tres {
    public int test() { return 4; }
}

class Prueba {
    public static void main(String [] args) {
        Uno    obj3 = new Tres();
        Tres    obj4 = new Cuatro();
        System.out.println("obj3.test()          = " + obj3.test());
        System.out.println("obj3.resultado2()     = " + obj3.resultado2());
        System.out.println("obj3.resultado3()     = " + obj3.resultado3());
        System.out.println("obj4.resultado1()     = " + obj4.resultado1());
        System.out.println("obj4.resultado2()     = " + obj4.resultado2());
        System.out.println("obj4.resultado3()     = " + obj4.resultado3());
    }
}

```

SALIDA:

```

obj3.test()          = 3
obj3.resultado3()    =
obj3.resultado2()    =
obj4.resultado1()    = 4
obj4.resultado2()    = 4
obj4.resultado3()    = 2

```

# Contenido

- Introducción histórica
- Programas y paquetes
- Clases y objetos
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores. Excepciones
- Cadenas de caracteres
- Arrays
- Listas
- Herencia y redefinición de comportamiento
- Polimorfismo y vinculación **dinámica**
- **Clases abstractas e interfaces**



# Promoviendo subclasses: Clases abstractas

- Se puede definir una clase como resultado de una abstracción sobre otras clases recogiendo un estado y un comportamiento básicos, aunque no tenga sentido modelar objetos propios de la abstracción.
- Se etiquetan como **abstract** y pueden tener métodos sin definición, también etiquetados como **abstract**.
- Se utilizan para formar jerarquías.
- Se pueden utilizar como tipos, pero no se pueden crear instancias suyas.
- Deben tener subclasses que no sean abstractas para generar objetos.

```
abstract class Polígono {  
    protected Punto[] vértices;  
    ...  
    public void trasladar(double a, double b) {  
        for (int i = 0; i < vértices.length; i++) {  
            vértices[i].trasladar(a, b);  
        }  
    }  
    ...  
    abstract public double área();  
};
```

Polígono pol = ~~new Polígono();~~

```

abstract class Polígono {
    protected Punto[] vért;

    public Polígono(Punto[] vs) {
        vért = vs;
    }
    public void trasladar(double a, double b) {
        for(Punto pto : vért) pto.trasladar(a,b);
    }
    public double perímetro() {
        Punto ant = vért[0];
        double res = 0;
        for(Punto pto : vért) {
            res += pto.distancia(ant);
            ant = pto;
        }
        return res;
    }

    abstract public double área(); // No sabemos calcularla
}

```

```

class Triángulo extends Polígono {

    public Triángulo(...) {...} //
Constructor
    public double área() {
        return base()*altura() / 2;
    }
    public double base() {
        return vért[1].distancia(vért[2]);
    }
    public double altura() {
        Recta r = new
Recta(vért[1],vért[2]);
        return vért[0].distancia(r);
    }
    public String toString() {...}
}

```

```

class Rectangulo extends Polígono {

    public Rectangulo(...) {...} //
Constructor
    public double área() {
        return base()*altura();
    }
    public double base() {
        return vért[0].distancia(vért[1]);
    }
    public double altura() {
        return vért[1].distancia(vért[2]);
    }
    public String toString() {...}
}

```

# Interfaces

- Define un **protocolo de comportamiento**, es decir un **contrato** que las clases deberán respetar.
  - Las clases pueden implementar la interfaz respetando el contrato.
  - Se utilizarán cuando se demande el contrato.
- Una interfaz **sólo puede ser *extendida*** por otra interfaz.
- Una interfaz **puede heredar** de varias interfaces.
- Una clase **puede *implementar*** varias interfaces.

# Definición de interfaces

- En una interfaz sólo se permiten constantes, métodos abstractos y **métodos por defecto** (1.8).

`public static final`

`package, en caso de omisión`

```
public interface miInterfaz
    extends interfaz1, interfaz2 {
    String CAD1 = "SUN";
    String CAD2 = "PC";
    void valorCambiado(String producto, int val);
    ...
    default void valorPorDefecto() {...}
}
```

`public abstract`

# Implementación de interfaces

- Cuando una clase implementa una interfaz,
  - se **adhiera** al contrato definido en la interfaz y sus interfaces ascendentes. Esto significa que:
    - La clase **debe implementar** todos los métodos declarados en la interfaz.
    - Si algún método **no se implementa**, se considera como **abstracto**, y la clase debe etiquetarse como **abstract**.
  - **hereda** todas las constantes definidas en la jerarquía,

```
public class MiClase  
    extends Superclase1  
    implements Interfaz1, Interfaz2, ... {  
    ...  
}
```

```
interface FiguraCerrada {  
    void trasladar(double a, double b);  
    double perímetro();  
    double área();  
}
```

```
abstract class Polígono implements FiguraCerrada
```

```
{  
    protected Punto[] vért;  
  
    public Polígono(Punto[] vs) {  
        vért = vs;  
    }  
    public void trasladar(double a, double b) {  
        for(Punto pto : vért) pto.trasladar(a,b);  
    }  
    public double perímetro() {  
        Punto ant = vért[0];  
        double res = 0;  
        for(Punto pto : vért) {  
            res + = pto.distancia(ant);  
            ant = pto;  
        }  
        return res;  
    }  
  
    abstract public double área();  
    // No sabemos calcularla  
}
```

```
class Círculo implements FiguraCerrada {  
    protected Punto centro;  
    protected double radio;  
  
    public Polígono(Punto ctro, double r ) {  
        centro = ctro;  
        radio = r;  
    }  
    public void trasladar(double a, double b) {  
        centro.trasladar(a,b);  
    }  
    public double perímetro() {  
        return 2 * Math.PI * radio;  
    }  
    abstract public double área() {  
        return Math.PI * radio * radio;  
    }  
}
```

# Uso de interfaces

- Se pueden declarar variables y parámetros de tipo una interfaz.
- Se requieren instancias de clases que implementen la interfaz.

```
FiguraCerrada figura;  
figura = new Círculo(new Punto(0,0),1);
```

Declaración de  
una variable de  
tipo interfaz

```
figura = new Triángulo(  
    new Punto(0,0),  
    new Punto(1,1),  
    new Punto(2,2) );
```

Objeto de la  
clase Elipse, que  
implementa la  
interfaz

Objeto de la  
clase Triángulo,  
que implementa  
la interfaz, pues  
hereda de la  
clase Polígono