

# Introducción a Java



## Contenido

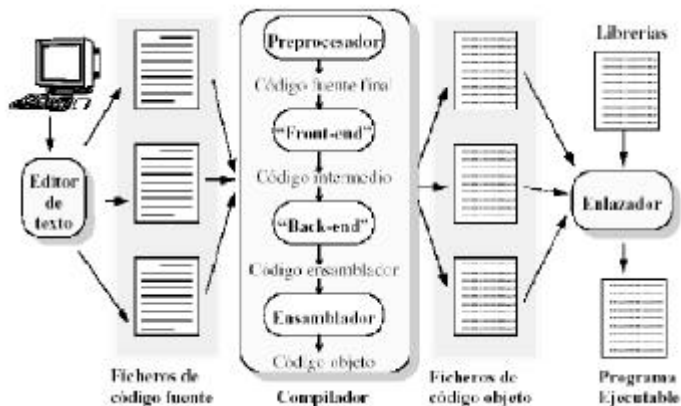
- Introducción histórica
- Programas y paquetes
- Clases y objetos
- Elementos del lenguaje:
  - Expresiones
  - Operadores
  - Instrucciones
  - Bloques
- Control de errores
- Cadenas de caracteres
- Arrays
- Herencia
- Polimorfismo y vinculación dinámica
- Clases abstractas e interfaces

# Introducción a Java

- Desarrollado por Sun. Aparece en 1995
- Basado en C++ (y algo en Smalltalk) eliminando
  - definiciones de tipos de valores y macros,
  - punteros y aritmética de punteros,
  - necesidad de liberar memoria.
- 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"
  - ficheros fuente `.java` se convierten en ficheros *bytecode* `.class`
- Interpretado
  - ficheros `.class` son interpretados por la máquina virtual de Java (JVM)

3

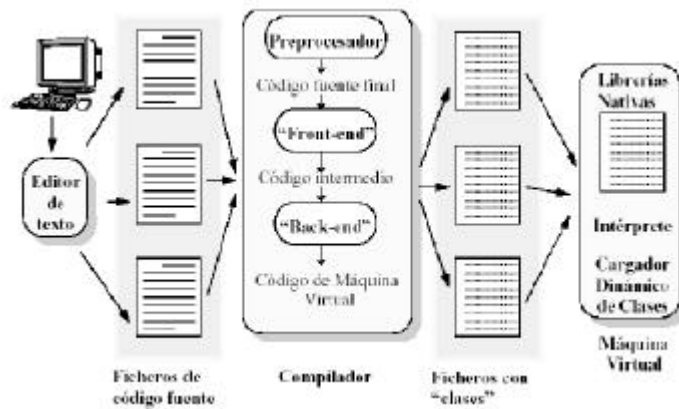
# Introducción a Java



Compilación y Enlazado (Ej. C++)

4

# Introducción a Java



## Compilación e Interpretación (Java)

5

# Programa en Java

- Conjunto de clases diseñadas para colaborar en una tarea, con una clase (pública) distinguida que contiene un método de clase:

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

que desencadena la ejecución del programa.

- Las demás clases pueden estar definidas *ad hoc* o pertenecer a una biblioteca de clases.

6

# 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 compilarse 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)

7

# Ejecución de un programa

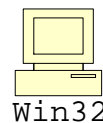
```
public class HolaMundo {  
    public static void main(String[] args) {  
        System.out.println("Hola Mundo");  
    }  
}
```

HolaMundo.java

**javac**

Bytecodes  
HolaMundo.class

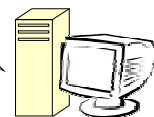
**java**



Win32



MacOS



Solaris

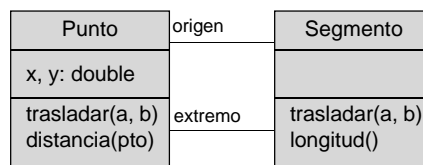
8

# Paquetes

- En Java las clases se organizan en **paquetes** (**package**): mecanismos lógicos de organización para agrupar clases relacionadas.
  - Si una clase no especifica el paquete al que pertenece, formará parte del paquete por defecto (sin nombre)
  - Las clases de un paquete sólo se pueden acceder por sus nombres desde otra clase dentro del mismo paquete
  - Para acceder a ellas desde otro paquete hay que hacerlo precediéndolas con el nombre del paquete o utilizando la construcción **import**

9

# Paquetes



10

# Paquetes

```
package pacl;  
  
public class Punto {  
    private double x, y;  
  
    public Punto() { x = y = 0; }  
    public Punto(double a, double b) { x = a; y = b; }  
    public double abscisa() { return x; }  
    public double ordenada() { return y; }  
    ...  
}
```

Punto.java

11

# Paquetes

```
package pacl;  
  
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);  
    }  
    ...  
}
```

mismo paquete

correcto

Segmento.java

12

# Paquetes

package **pac2;** — **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

13

# Paquetes

package pac2;

```
public class segmento {  
    private pac1.Punto origen, extremo;  
    public Segmento(double x1, double y1, double x2, double y2) {  
        origen = new pac1.Punto(x1, y1);  
        extremo = new pac1.Punto(x2, y2);  
    }  
    ...  
}
```

**correcto**

Segmento.java

14

# Paquetes

```
package pac2;  
import pac1.Punto; // import pac1.*;  
  
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

15

# 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
- ...

16



# Acceso a las bibliotecas de Java

- A las clases incluidas en `java.lang` se puede acceder simplemente por sus nombres sin necesidad de anteponer (o importar) el nombre del paquete, p.e.: `System` o `Math`.
- Para el resto de clases de las bibliotecas sí hay que especificar el nombre del paquete

17

## 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);
    }
}
```

18

# Ejemplo

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

19

# Paquetes

- Todas las clases de un paquete deben estar localizadas en un mismo subdirectorio (o carpeta).
- Los paquetes del sistema cuelgan de varios subdirectorios específicos:
  - .../java
  - .../javax
- Por defecto el compilador busca un paquete necesario entre los del sistema y en el directorio raíz del proyecto
- Para que busque paquetes en otro sitio hay que indicárselo en la variable del entorno **CLASSPATH**

20

# Paquetes

- La variable del entorno **CLASSPATH** contiene una lista con todos directorios de búsqueda de los paquetes.
- 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.

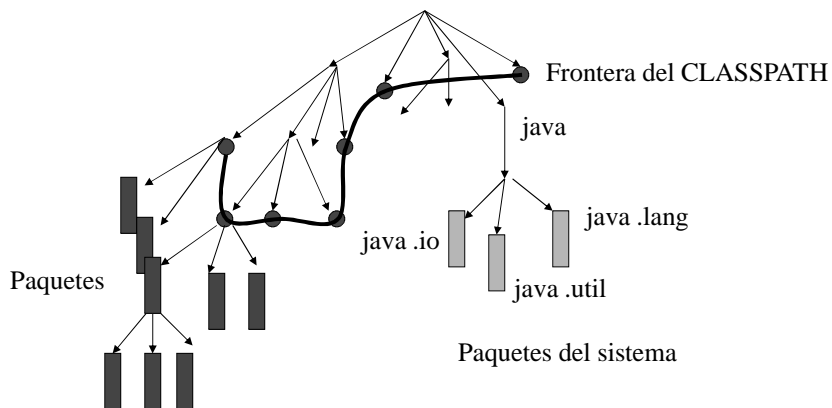
Ejemplos:

`java.util`

`java.awt.event`

21

# Paquetes



22

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

23

# 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(this.distancia(part), 2);
    }
}
```

Cabecera  
Cte. de clase

Var. de instancia

Constructor

Método de instancia

24

# Métodos y variables de instancia

- Las variables de instancia (o atributos) se acceden etiquetándolas con el nombre o la referencia de la instancia.
- Cada instancia tiene sus propias variables de instancia.
  - Un punto tiene dos variables de instancia x, y mientras que una partícula tiene tres (x, y, m)
- Los métodos de instancia se invocan mediante mensajes contruidos precediendo el nombre del método con el nombre o la referencia de la instancia.
- Un método de instancia tiene acceso a las variables de clase y a las variables de instancia propias (del receptor).

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

25

# Métodos y variables de clase

- Las variables de clase
  - Son comunes a todos los objetos de la clase.
  - Se declaran como **static**.
  - Se acceden/invocan etiquetando sus nombres con el nombre de la clase cuando son visibles (y también con el nombre de alguna instancia).
- Los métodos de clase
  - Sólo tienen acceso a las variables de clase.
  - Se declaran como **static**.
  - Se acceden/invocan etiquetando sus nombres con el nombre de la clase cuando son visibles (y también con el nombre de alguna instancia).

```
public double atracción(Partícula part) {  
    return G * masa * part.masa /  
           Math.pow(distancia(part), 2);  
}
```

26

# Métodos y variables de clase

```
class Vuelo {  
    static private int sigVuelo = 1;  
    private String localizadorVuelo;  
  
    static public int nuevoIdentificador() {  
        vuelo = sigVuelo;  
        sigVuelo++;  
        return vuelo;  
    }  
  
    public Vuelo(String lin) {  
        localizadorVuelo = lin + "_" + nuevoIdentificador();  
    }  
    ...  
}  
  
Vuelo v1 = new Vuelo("Iberia");           // Iberia_1  
Vuelo v2 = new Vuelo("Lufhtansa");       // Lufhtansa_2  
Vuelo v3 = new Vuelo("Iberia");          // Iberia_3
```

27

# Métodos y variables de instancia

```
class Ejemplo {  
    // Variable de clase  
    static public int a;  
    // Variable de instancia  
    public int b;  
    // método de instancia  
    public int getSuma() {  
        // tiene acceso a ambas  
        return a + b;  
    }  
    ...  
}
```

28

# Métodos (parámetros)





- En Java todos los parámetros son por valor
- No existe el paso por referencia (como existe por ejemplo en C++)

29

# 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 heredadas
- **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>protected</b>	SÍ	SÍ	SÍ	NO
	+	<b>public</b>	SÍ	SÍ	SÍ	SÍ
	~	<b>package</b>	SÍ	SÍ	NO	NO

30

## La vida de los objetos

- Los objetos son las instancias de las clases.
- Durante la ejecución de un programa típico de Java se van creando objetos que interactúan entre sí, enviándose mensajes, para realizar determinadas tareas.
- Una vez que los objetos completan su trabajo se van eliminando de forma automática liberando los recursos ocupados.

31

## Creación de objetos

- Los objetos hay que crearlos
  - Se le reserva de espacio de memoria
  - Se le asigna unos valores iniciales a sus variables de estado
  - Se debe utilizar un constructor:

```
new <constructor>(<lista args>)
```

- **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));
```

32



# Constructores de objetos

- Operaciones definidas en las clases con el mismo nombre de la clase que no devuelven valor.
  - Habilitan espacio para un objeto de la clase.
  - Establecen valores iniciales para las variables de estado.
- Una clase puede definir varios constructores (con distinto número de argumentos o con argumentos de distintos tipos) o ninguno (en cuyo caso se aplica uno por defecto sin argumentos).
  - Si no hay constructores entonces existe el “por defecto”
  - Si hay constructores, el “por defecto no se crea”

33

# Clases en Java

```
class Punto {
    private double x, y;
    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 double distancia(Punto pto) {
        return Math.sqrt(Math.pow(x - pto.x, 2)
            + Math.pow(y - pto.y, 2));
    }
}
Punto = pto;
pto = new Punto(); //x=0, y=0;
//Constructor por defecto
```

34

# Variables de objeto

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

```
Punto pto;
```

**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);
```

- Ya puede recibir mensajes.

- Estos dos pasos se pueden realizar simultáneamente:

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

35

# Uso de objetos

- A las variables de estado de un objeto se accede de la forma **pto.x**, siempre que sea visible. (desde fuera de la clase)

- A las de instancia dentro de un método con **this.x**

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

- Si no hay conflicto de nombres, **this** puede suprimirse.

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

- 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 aconsejable”.*

36

# Ejemplo

```
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(this.distancia(part), 2);
    }
    public double atracción(Partícula part) {
        return G * masa * part.masa /
            Math.pow(distancia(part), 2);
    }
}
```

37

# Ejemplo(2)

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

38

# Uso de objetos

- Invocación de los métodos
  - Los métodos cuando son visibles, se invocan mediante la forma:

```
Punto pto = new Punto(3,4);  
pto.trasladar(2, 2);
```
  - 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.

39

# Asignación y Copia

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

40

## Uso de objetos (asignación)

```
class Segmento {
    private Punto origen, extremo;
    public Segmento(Punto pt1, Punto pt2) {
        origen = pt1;
        extremo = pt2;
    }
    ... // Otros métodos
    public float longitud() {
        return origen.distancia(extremo);
    }
}

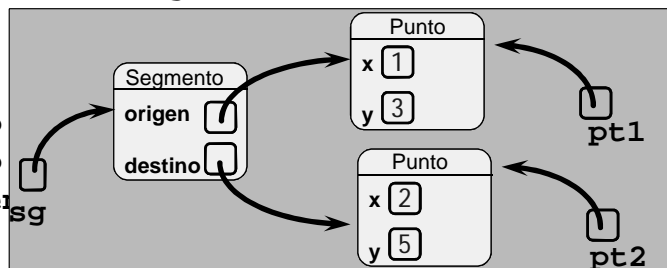
Punto pt1 = new Punto(1,3);
Punto pt2 = new Punto(2,5);
Segmento sg = new Segmento(pt1,pt2);
```

41

## Uso de objetos (asignación)

```
class Segmento {
    private Punto origen, extremo;
    public Segmento(Punto pt1, Punto pt2) {
        origen = pt1;
        extremo = pt2;
    }
    ... // Otros métodos
    public float longitud() {
        return origen.distancia(extremo);
    }
}
```

Punto  
Punto  
Segmento



42

## Uso de objetos (duplicación)

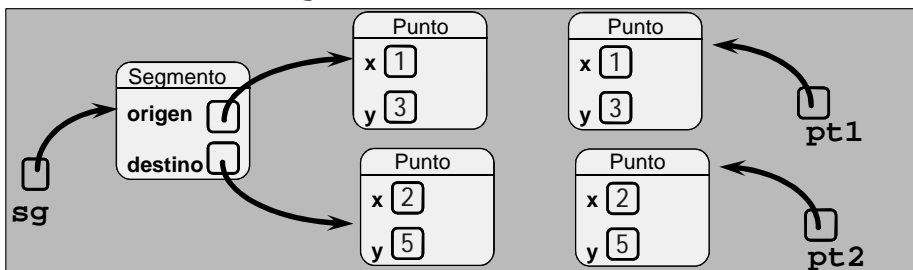
```
class Segmento {
    private Punto origen, extremo;
    public Segmento(Punto pto1, Punto pto2) {
        origen = new Punto(pto1.x(), pto1.y());
        extremo = new Punto(pto2.x(), pto2.y());
    }
    ... // Otros métodos
    public float longitud() {
        return origen.distancia(extremo);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

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

43

## Uso de objetos (duplicación)

```
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
  - Automática cuando el objeto es inalcanzable
    - Porque se pierden todas las referencias a él
    - Se puede provocar la pérdida de una referencia a un objeto  
`pto = null;`
- Se puede solicitar la eliminación automática
  - invocando el método de clase `gc ( )` de la clase `System`.

45

# Tipos y valores

- En Java hay tipos básicos.
  - Llamamos valores a los datos de estos tipos.
- Sólo existen los siguientes tipos básicos:

<code>byte</code> (entero de 8 bits)	<code>short</code> (entero de 16 bits)
<code>int</code> (entero de 32 bits)	<code>long</code> (entero de 64 bits)
<code>float</code> (decimal de 32 bits)	<code>double</code> (decimal de 64 bits)
<code>char</code> (Unicode de 16 bits)	<code>boolean</code> ( <code>true</code> , <code>false</code> )
- El número de bits dedicado es independiente de las plataformas sobre las que se ejecuten los programas.
- No se pueden definir más tipos básicos.

46

# Tipos y valores

- Los valores, al igual que los objetos, se pueden manipular mediante variables.
- Las variables de tipos básicos contienen a los valores, no los referencian.
- Los valores también se pueden manipular directamente, usando sus representaciones literales

```
int    56, 0123, 0xaf, 1_000_000 (java 1.7)
long   84561, 33456L
double 4.66, 23.7e3, 35.213d, 23.2D
float  67.345f, 34.122F, 21.34e2f
char   'a', '$', '\t', '\u0061'
boolean true, false

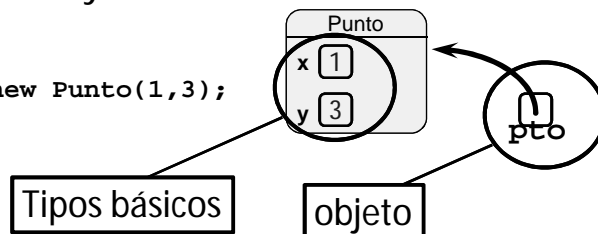
int 1_000_000 0b1001_1100 (desde Java1.7)
```

47

# Tipos básicos versus 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);
```



48



# Conversiones de tipos y clases

- En Java se producen conversiones de tipo o de clase de forma **implícita** en ciertos contextos.

Estas conversiones se producen 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.

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

```
(<tipo/clase>) <expresión>
```

Sólo se comprueban durante la ejecución.

49

# 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 actuales 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)

50

# 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;`

51

# Constantes

- Una variable se puede declarar como constante precediendo su declaración con la etiqueta **final**:  
`final int varFinal = 0;`
- La inicialización de una variable final se puede hacer en cualquier momento posterior a su declaración.
- Cualquier intento de cambiar el valor de una variable final después de su inicialización produce un error en tiempo de compilación.

52

# 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 no se separan y comienzan con mayúscula.

```
long valorMáximo;
```

- Nombres de clase igual, pero comenzando con mayúscula.

```
class PilaNat ...
```

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

```
final int CTE_GRAVITACIÓN
```

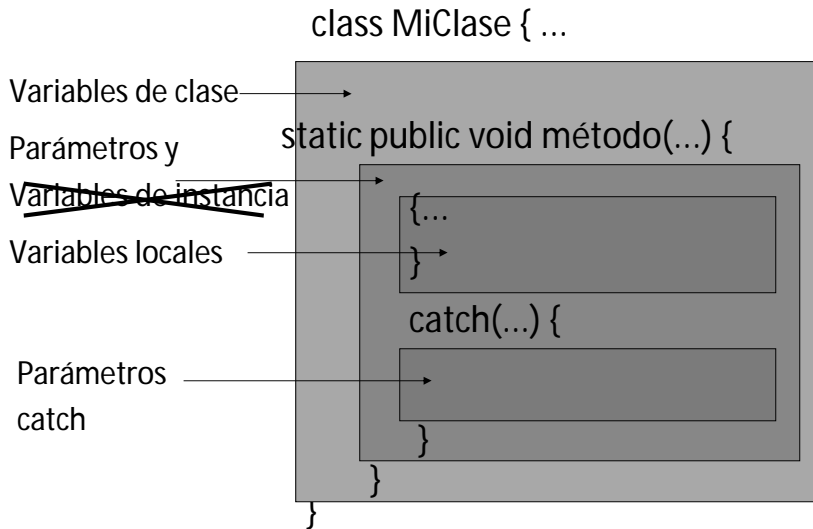
53

# 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 calificar.
- 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

54

# Ámbitos



55

## Inicialización de variables

- Cuando no se les asigna un valor explícitamente
  - 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.
- Valores de inicialización automática:

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

56

# 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.

57

# Operadores (I)

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

58

## 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.

`3 * 5 + x / 7.3`            `y *= x += 7.3`

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

59

## 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>new</code>	<code>(tipo)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

60

# 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
```

61

# 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

62

## Sentencias de declaración

- Las sentencias 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;
```

63

## 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

64



# 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.

65

# 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>
```

66

## Sentencias de selección (II)

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

67

## Bloques

- Un bloque es un grupo de cero o más sentencias, encerradas entre llaves, dando lugar a una sentencia compuesta.

```
{ <sentencia1>  
  ...  
  <sentenciaN>  
}
```

- Un bloque se puede usar en cualquier parte donde se necesite una sentencia simple.

68

## Control de excepciones (I)

- Java dispone de un mecanismo de ayuda para la comunicación y el manejo de errores conocido como *control de excepciones*.
- Cuando se produce un error en un método:
  1. se genera un objeto de la clase **Exception**, o de alguna heredera, con información sobre el error,
  2. se interrumpe el flujo normal de ejecución, y
  3. el entorno de ejecución trata de encontrar un tratamiento para dicho objeto dentro del propio método o en uno anterior en la pila de activaciones.

69

## Control de excepciones (II)

Existen tres sentencias relacionadas con el control de excepciones:

- **try** que delimita un bloque de instrucciones donde se puede producir una excepción,
- **catch** que identifica un bloque de código asociado a un bloque **try** donde se trata un tipo particular de excepción,
- **finally** que identifica un bloque de código que se ejecutará después de un bloque **try** con independencia de que se produzcan o no excepciones.

70

## 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>
}
```

71

## Cadenas de caracteres

- Las cadenas de caracteres se representan en Java como secuencias de caracteres Unicode encerradas entre comillas dobles:

“Ejemplo de cadena de caracteres”

- La forma más directa de definir cadenas de caracteres es mediante la clase `String`:

```
String cadena = "hola";
cadena = "adiós";
cadena = new String("hola y adiós");
cadena = new String(); // cadena vacía
```

72

# 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)
- Las cadenas (String) en Java son objetos inmutables
- Si se intenta acceder a una posición no válida el sistema lanza una excepción:
  - StringIndexOutOfBoundsException

73

# 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 primitivos ...

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

SALIDA:                p es Punto@119c0982

74

# 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)
```

75

# Cadenas de caracteres

- Comparación de cadenas:
  - `c1.equals(c2)` – devuelve true si c1 y c2 son iguales y false en otro caso.
  - `c1.compareTo(c2)` – devuelve un entero menor, igual o mayor que cero cuando c1 es menor, igual o mayor que c2.
- ¡ojo!
  - `c1 == c2` compara variables referencia
- Otros métodos:
  - `indexOf(...)` – primera posición de ... (carácter o cadena)
  - `lastIndexOf(...)` – última posición de ... (carácter o cadena)
  - `toLowerCase()` – nuevo String con todo minúsculas
  - `toUpperCase()` – nuevo String con todo mayúsculas
  - `substring(inicio,fin)` – nuevo String con caracteres a partir de posición inicio y hasta posición fin (sin incluirlo)

76

# 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 una variable de instancia **length** que sólo se puede consultar.
- Los elementos de un array se inicializan con los valores por defecto (excepto al usar arrays literales)

77

# Arrays en Java (II)

- 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());  
}
```

78

## Array de arrays (I)

- Las componentes de un array pueden ser arrays.
- Al crear un array multidimensional con el operador **new** solo hace falta fijar el tamaño de la primera dimensión.

```
int[][] matriz = new int[4][];
for (int i = 0; i < matriz.length; i++) {
    matriz[i] = new int[i + 5];
    for (int j = 0; j < matriz[i].length; j++) {
        matriz[i][j] = i + j;
    }
}
```

79

## Array de arrays (II)

```
String[][] familias = {
    {"Picapiedra", "Pedro", "Wilma", "Pebbles", "Dino"},
    {"Mármol", "Pablo", "Betty", "Bam Bam"},
    {"Simpson", "Homer", "Marge", "Burt", "Lisa", "Maggie"}
};

for (int i = 0; i < familias.length; i++) {
    System.out.print(familias[i][0] + ": ");
    for (int j = 1; j < familias[i].length; j++) {
        System.out.print(familias[i][j] + " ");
    }
    System.out.println();
}
```

80



## 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)
```

81

## Copia de arrays (II)

```
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);
```

82

# Recorridos en 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);  
}
```

83

# Mostrar un array

El siguiente segmento de código

```
int [] ar = {1,3,5,7,9};  
System.out.println(java.util.Arrays.toString(ar));
```

mostrará en la salida estándar

```
[1, 3, 5, 7, 9]
```

- La clase **Arrays** está en **java.util** y se estudiará más adelante. Tiene muchas más funcionalidades como algoritmos de búsqueda, ordenación, etc.

84

## Datos enumerados: `enum`

```
enum Semana {Lun,Mar,Mie,Jue,Vie,Sab,Dom};

class Ej {
    public static void main(String [] args) {
        Semana s = Semana.Lun;
        Semana t = Semana.valueOf("Mie");

        for(Semana se : Semana.values()) {
            System.out.print(se + " ");
        }
    }
}
```

Lun Mar Mie Jue Vie Sab Dom

85

## Clases anidadas

- Se definen dentro del cuerpo de otra clase.
- Aunque se pueden distinguir diversos tipos de clases anidadas (internas, locales, anónimas), dependiendo del ámbito en el que se declaren, solo consideraremos las denominadas:
  - **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 debe cualificarse con el nombre de la clase externa (si es visible).

86

# 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;  
    }  
    ...
```

87

# Subclases/Herencia

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

```
class miClase extends superclase {  
    ...  
}
```
- En Java sólo se permite *herencia simple*, por lo que pueden establecerse jerarquías de clases.
- Todas las jerarquías confluyen en la clase **Object** de **java.lang** que recoge los comportamientos básicos que debe presentar cualquier clase.

88

# 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 Punto()  
    masa = 0;  
}
```

89

# Herencia, variables y métodos

- Métodos de instancia:
  - Un método de instancia de una clase puede redefinirse en una subclase.
    - Salvo si el método está declarado como `final` (o la clase).
  - 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 al redefinido, 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.

90

```

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

```

91

```

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

```

92

```

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

```

93

## Prohibiendo subclases

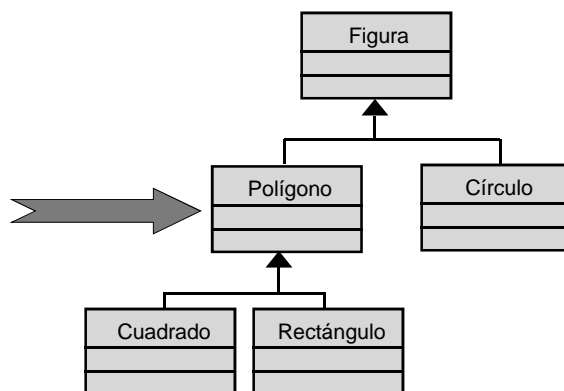
- Por razones de seguridad (una subclase puede sustituir a su superclase donde ésta sea necesaria y tener comportamientos muy distintos) o de diseño, se puede prohibir la definición de subclases para una clase etiquetándola con **final**.
- El compilador rechazará cualquier intento de definir una subclase para una clase etiquetada con **final**.
- También se pueden etiquetar con **final**:
  - métodos, para evitar su redefinición en alguna posible subclase, y
  - variables, para mantener constantes sus valores o referencias.

94

# Promoviendo subclasses: Clases abstractas

- En Java 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.
- Estas clases se etiquetan como **abstract** y pueden tener métodos sin definición, también etiquetados como **abstract**.
- Con estas clases se pueden formar jerarquías.
- Estas clases se pueden utilizar como tipos, pero no se pueden crear instancias suyas.
- Deben tener subclasses que no sean abstractas para generar objetos.

95





```

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 Cuadrado extends Polígono {
    public Cuadrado(...) {...}
    public double área() {
        return lado()*lado();
    }
    public double lado() {
        return vért[0].distancia(vért[1]);
    }
    public String toString() {...}
}

```

```

class Rectángulo extends Polígono {
    public Rectángulo(...) {...}
    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

- Una interfaz define un protocolo de comportamiento que debe ser implementado por cualquier clase que pretenda ofrecer ese comportamiento.
- Una interfaz sólo puede ser *extendida* por otra interfaz.
- Una clase puede *implementar* varias interfaces.

99

## Definición de interfaces

- En una interfaz sólo se permiten constantes y métodos abstractos.

```
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);
    ...
}
```

public abstract

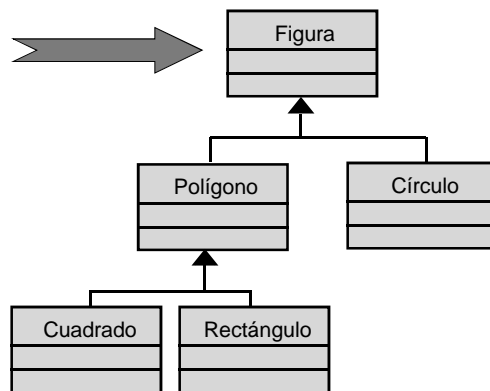
100

# Implementación de interfaces

- Cuando una clase implementa una interfaz,
  - se adhiere al protocolo definido en la interfaz y en sus superinterfaces,
  - *hereda* todas las constantes definidas en la jerarquía,
  - *debe implementar* todos los métodos, salvo que sea una clase que se quiera mantener abstracta (en cuyo caso, los métodos no implementados aparecerán como **abstract**).

```
public class miClase
    extends superclase1
    implements interfaz1, interfaz2 {
    ...
}
```

101



```

interface Figura {
    double perímetro();
    double área();
}

```

```

abstract class Polígono
    implements Figura{

    protected Punto[] vért;

    public Polígono(Punto[] vs) {
        vért = vs;
    }

    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();
}

```

```

class Círculo implements Figura{

    private double radio;

    public Círculo(double r) {
        radio = r;
    }

    public double perímetro() {
        return 2.0 * Math.PI * radio;
    }

    public double área() {
        return Math.PI * radio * radio;
    }
}

```

103

```

abstract class Polígono implements Figura {
    protected Punto[] vért;

    public Polígono(Punto[] vs) {
        vért = vs;
    }

    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();
}

```

```

class Cuadrado extends Polígono {

    public Cuadrado(...) {...}
    public double área() {
        return lado()*lado();
    }
    public double lado() {
        return vért[0].distancia(vért[1]);
    }
    public String toString() {...}
}

```

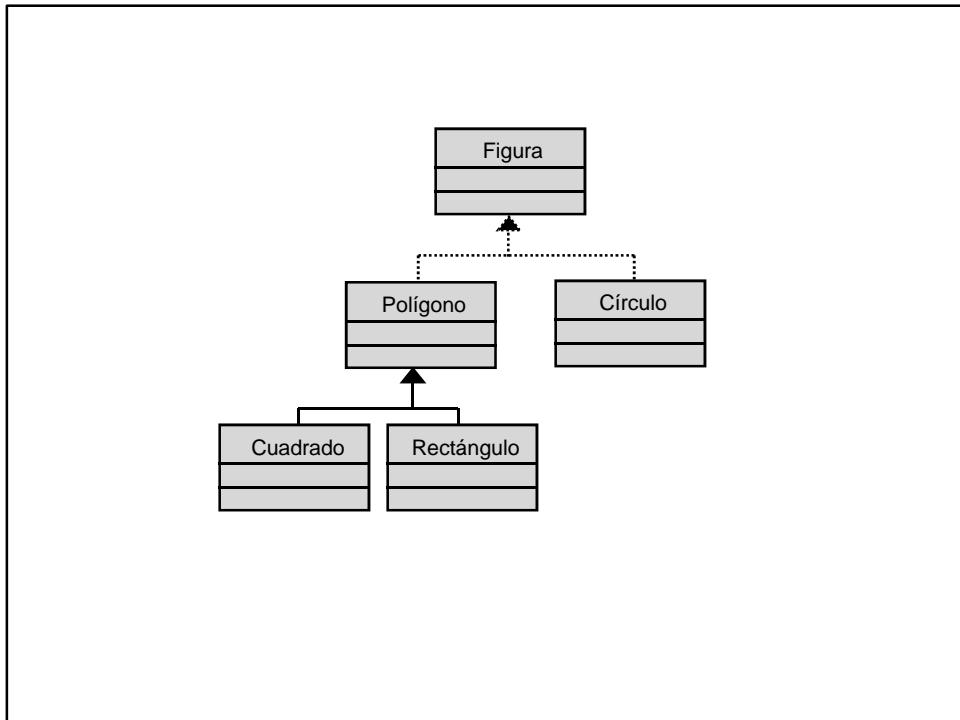
```

class Rectángulo extends Polígono {

    public Rectángulo(...) {...}
    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() {...}
}

```

104



## Uso de interfaces

- No se pueden crear instancias de una interfaz.
- Podemos declarar variables del tipo definido por una interfaz, que pueden hacer referencia a cualquier objeto de una clase que implementa la interfaz (y de las herederas de ellas).