

Clases y objetos en Java

Definición de una clase: estructura de datos + funciones

```
class CuentaBancaria {  
    long numero;  
    String titular;  
    long saldo;  
    void ingresar (long cantidad) {  
        saldo += cantidad;  
    }  
    void retirar (long cantidad) {  
        if (cantidad > saldo)  
            System.out.println ("Saldo insuficiente");  
        else saldo -= cantidad;  
    }  
}
```

} Variables

} Métodos

2

Variables

- Definen una estructura de datos (como un struct de C)

```
class CuentaBancaria {      struct CuentaBancaria {
    long numero;             long numero;
    String titular;         char *titular;
    long saldo;             long saldo;
}                            };
```

- Pueden almacenar objetos

```
class CuentaBancaria {      class Cliente {
    long numero;             String nombre;
    Cliente titular;         long dni;
    long saldo = 0;         }
}
```

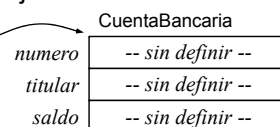
- Por defecto se inicializan a 0 ó null

Las variables se pueden inicializar directamente en la clase (con una expresión arbitraria que no genere checked exceptions)

3

Creación de objetos

- Una clase define un tipo de dato que se puede utilizar para declarar variables
`CuentaBancaria cuenta1, cuenta2;`
- Declarar un objeto es declarar una referencia a un objeto
- Los objetos se crean con el operador `new`
`cuenta1 = new CuentaBancaria ();`
- Crear un objeto significa reservar espacio en memoria para sus variables
- `new` reserva memoria para un objeto y devuelve una referencia al objeto

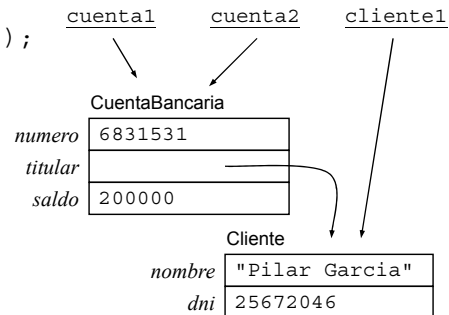


Los objetos siempre utilizan memoria dinámica

4

Creación de objetos y acceso a variables

```
CuentaBancaria cuenta1, cuenta2;  
cuenta1 = new CuentaBancaria ();  
cuenta2 = cuenta1;  
Cliente cliente1 = new Cliente ();  
  
cliente1.nombre = "Luis Gomez";  
cliente1.dni = 25672046;  
cuenta1.numero = 6831531;  
cuenta1.titular = cliente1;  
cuenta1.saldo = 100000;  
cuenta2.saldo = 200000;  
  
cuenta2.titular.nombre = "Pilar Garcia";
```



5

Métodos

- Los métodos son funciones definidas dentro de una clase (Similar a puntero a función en un struct de C)
- Los métodos pueden referenciar directamente a las variables de la clase
- Los métodos se invocan sobre un objeto de la clase a la que pertenecen
- Al ejecutar un método invocado sobre un objeto de clase A, las variables de la clase A toman el valor que tienen en el objeto

```
cuenta2.ingresar(1000);
```

```
numero ← cuenta2.numero  
titular ← cuenta2.titular  
saldo ← cuenta2.saldo  
  
void ingresar (long cantidad) {  
    saldo += cantidad;  
}
```

`cuenta2.saldo`

6

Llamadas a métodos desde un método

- Los métodos pueden invocar directamente otros métodos de la misma clase
- Al ejecutar un método invocado sobre un objeto de clase A, las llamadas a otros métodos de la clase A se ejecutan sobre el mismo objeto a menos que se invoquen sobre otro objeto

```
cuenta3 = new CuentaBancaria ();  
cuenta2.transferencia (cuenta3, 1000);
```

```
class CuentaBancaria {  
    ...  
    void transferencia (CuentaBancaria destino, long cantidad) {  
        if (cantidad <= saldo ) {  
            retirar (cantidad);  
            destino.ingresar (cantidad);  
        }  
    }  
}
```

cuenta2.retirar (cantidad)

Los métodos se ejecutan en el contexto de un objeto

- Objetos a los que puede acceder un método:
 1. Objeto de la invocación: implícitamente, accediendo directamente a sus variables y métodos
 2. Objeto definido en una variable local
 3. Objeto pasado como argumento
 4. Objeto almacenado en una variable de la clase
- En C, el objeto de la invocación sería un argumento más
- En POO, el objeto de la invocación juega un papel distinto: el método invocado pertenece al objeto y no a la inversa
- En el código de un método, el objeto de la invocación no se ve explícitamente: se accede a sus variables y métodos

Objetos accedidos desde un método

```

class X { String nombre; }
class Y { int i; }
class Z { String nombre; }

class A {
    int num;
    X obj4;
    void f (int n, Y obj3) {
        Z obj2 = new Z ();
        obj4.nombre = obj2.nombre;
        num = obj3.i + n;
    }
}
    
```

1. Variable del objeto de la invocación
2. Objeto definido en variable local
3. Objeto pasado como argumento
4. Objeto almacenado en variable de clase

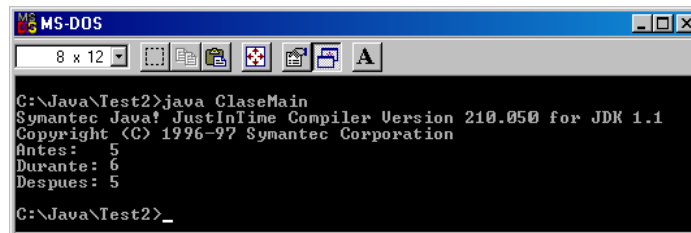
El objeto de la invocación (1) no se ve como los otros objetos (2, 3, 4) pero está implícito: el método `f` accede a sus variables

9

Paso de argumentos: siempre por valor

```

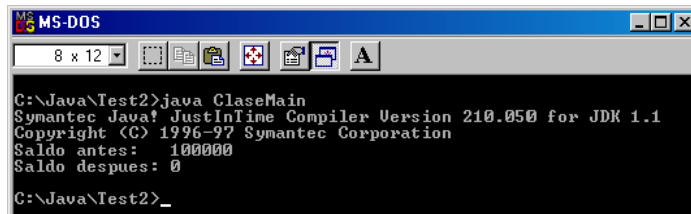
class ClaseMain {
    public static void main (String args[]) {
        int n = 5;
        System.out.println ("Antes: " + n);
        f (n);
        System.out.println ("Despues: " + n);
    }
    static void f (int i) { System.out.println ("Durante: " + ++i); }
}
    
```



10

Paso de argumentos: referencias (I)

```
class ClaseMain {
    public static void main (String args[]) {
        CuentaBancaria cuenta = new CuentaBancaria ();
        cuenta.saldo = 100000;
        System.out.println ("Saldo antes:  " + cuenta.saldo);
        arruinar (cuenta);
        System.out.println ("Saldo despues: " + cuenta.saldo);
    }
    static void arruinar (CuentaBancaria cnt) {
        cnt.saldo = 0;
        cnt = null;
    }
}
```

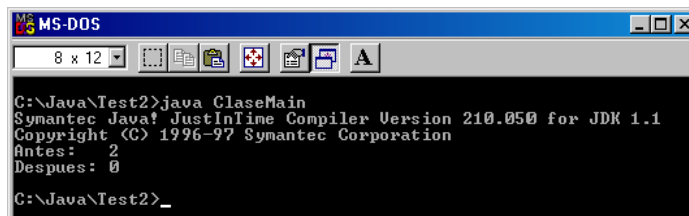


```
MS-DOS
8 x 12
C:\Java\Test2>java ClaseMain
Syntec Java! JustInTime Compiler Version 210.050 for JDK 1.1
Copyright (C) 1996-97 Syntec Corporation
Saldo antes: 100000
Saldo despues: 0
C:\Java\Test2>_
```

11

Paso de argumentos: referencias (II)

```
class ClaseMain {
    public static void main (String args[]) {
        int a[] = {5, 4, 3, 2, 1};
        System.out.println ("Antes:  " + a[3]);
        f (a);
        System.out.println ("Despues: " + a[3]);
    }
    static void f (int x[]) {
        x[3] = 0;
        x = new int[8];
        x[3] = 5;
    }
}
```



```
MS-DOS
8 x 12
C:\Java\Test2>java ClaseMain
Syntec Java! JustInTime Compiler Version 210.050 for JDK 1.1
Copyright (C) 1996-97 Syntec Corporation
Antes: 2
Despues: 0
C:\Java\Test2>_
```

12

Constructores

- “Métodos” que se ejecutan automáticamente al crear los objetos de una clase (i.e. al reservar memoria para los objetos, i.e. al invocar a `new`)
- Finalidad típica
 - Valores iniciales para las variables de los objetos
 - Otras operaciones de inicialización
- Utilidad
 - Simplificación de la sintaxis
 - Encapsulamiento de las variables de los objetos: evitar el acceso externo

13

Constructores: ejemplo

```
class Cliente {
    String nombre;
    long dni;
    Cliente (String str, long num) {
        nombre = str; dni = num;
    }
}

class CuentaBancaria {
    long numero;
    Cliente titular;
    long saldo;
    CuentaBancaria (long num, Cliente clt, long s) {
        numero = num; titular = clt; saldo = s;
    }
}
```

14

Creación de objetos con constructores

Los constructores se ejecutan automáticamente al crear los objetos

```
Cliente cliente1 = new Cliente ("Luis Gomez", 25672046);
```

| Cliente | |
|---------|--------------|
| nombre | "Luis Gomez" |
| dni | 25672046 |

```
CuentaBancaria cuenta1 =  
    new CuentaBancaria (6831531, cliente1, 100000);
```

```
CuentaBancaria cuenta2 =  
    new CuentaBancaria (8350284,  
                        new Cliente ("Pilar Garcia", 15165442),  
                        200000);
```

15

Constructores por defecto

- Si no se definen constructores, Java proporciona uno por defecto
- Si se define un constructor, el constructor por defecto no es definido

```
class A {  
    A () { }  
}
```

```
class Cliente {  
    ...  
    Cliente (String str, long num) { ... }  
}
```

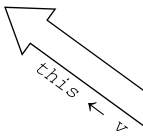
```
// Bloque main  
Cliente cliente1 = new Cliente ();  
// Error: No constructor matching Cliente() found in Cliente
```

16

La variable `this`

- Definida implícitamente en el cuerpo de los métodos
- Referencia al objeto sobre el que se invoca el método

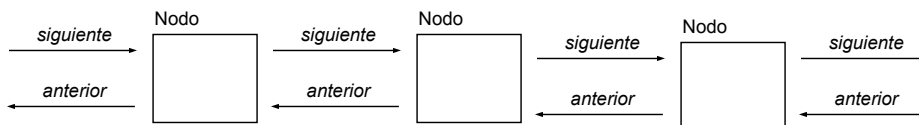
```
class Vector3D {
    double x, y, z;
    ...
    double productoEscalar (Vector3D u) {
        return x * u.x + y * u.y + z * u.z;
        // return this.x * u.x + this.y * u.y + this.z * u.z;
    }
    double modulo () {
        return (double) Math.sqrt (productoEscalar (this));
    }
}
```



```
// Bloque main
Vector3D v = new Vector3D (2, -2, 1);
v.modulo ();
```

La variable `this`: relaciones inversas

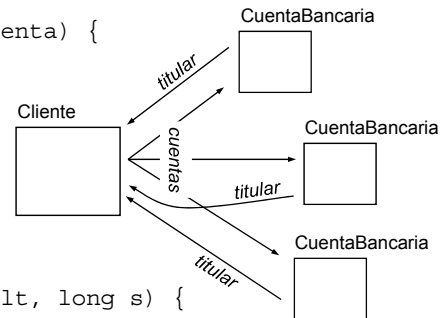
```
class Nodo {
    Nodo anterior;
    Nodo siguiente;
    void conectar (Nodo z) {
        siguiente = z;
        z.anterior = this;
    }
}
```



this en constructores

```
class Cliente {
    String nombre; long dni;
    Cliente (String str, long num) { nombre = str; dni = num; }
    CuentaBancaria cuentas[] = new CuentaBancaria [20];
    int ncuentas = 0;
    void nuevaCuenta (CuentaBancaria cuenta) {
        cuentas[ncuentas++] = cuenta;
    }
}
```

```
class CuentaBancaria {
    long numero;
    long titular;
    long saldo;
    CuentaBancaria (long num, Cliente clt, long s) {
        numero = num; titular = clt; saldo = s;
        clt.nuevaCuenta (this);
    }
}
```



19

Sobrecarga de métodos

```
class Plano3D {
    double a, b, c, d;
    Plano3D (double aa, double bb, double cc, double dd) {
        a = aa; b = bb; c = cc; d = dd;
    }
    boolean paralelo (Plano3D p) {
        Vector3D u = new Vector3D (a, b, c);
        Vector3D v = new Vector3D (p.a, p.b, p.c);
        return u.paralelo (v);
    }
    boolean paralelo (Recta3D r) {
        Vector3D u = new Vector3D (a, b, c);
        return u.perpendicular (r.vector);
    }
}
```

20

Sobrecarga de constructores

```

class Punto3D {
    double x, y, z;
    Punto3D (double xx, double yy, double zz) {
        x = xx; y = yy; z = zz;
    }
}

class Vector3D {
    double x, y, z;
    Vector3D (double xx, double yy, double zz) {
        x = xx; y = yy; z = zz;
    }
    Vector3D (Punto3D p, Punto3D q) {
        x = q.x - p.x; y = q.y - p.y; z = q.z - p.z;
    }
    boolean paralelo (Vector3D u) {
        return (x * u.y == y * u.x) && (x * u.z == z * u.x);
    }
    boolean perpendicular (Vector3D u) {
        return productoEscalar (u) == 0;
    }
}

```

21

Llamada a métodos sobrecargados

```

class Recta3D {
    Punto3D punto;
    Vector3D vector;
    Recta3D (Punto3D p, Vector3D v) {
        punto = p; vector = v;
    }
}

// Bloque main
Plano3D p1 = new Plano3D (2, 4, 3, 1);
Plano3D p2 = new Plano3D (1, 0, -2, 1);
Recta3D r = new Recta3D (new Punto3D (1, 0, 1),
                        new Vector3D (1, 1, -1));

p1.paralelo (p2);
p1.paralelo (r);

```

22

Sobrecarga de métodos: ambigüedad

```
class A {  
    void f (int n) {  
        System.out.println ("Tipo int");  
    }  
    void f (float x) {  
        System.out.println ("Tipo float");  
    }  
}
```

Se ejecuta la definición compatible más específica

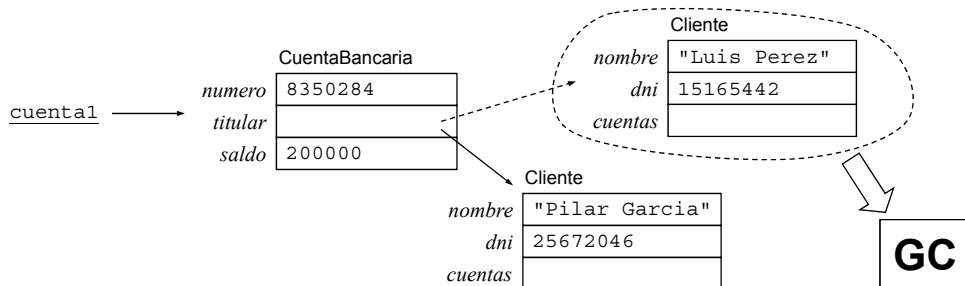
```
// Bloque main  
A a = new A();  
byte b = 3;  
long l = 3;  
double d = 3;  
a.f(l);  
a.f(b);  
a.f(d); // ERROR: necesita cast  
        // explícito
```

23

Destrucción de objetos (I)

- La memoria reservada para los objetos se libera por garbage collection (GC)
- GC libera la memoria de los objetos no referenciados en ninguna parte

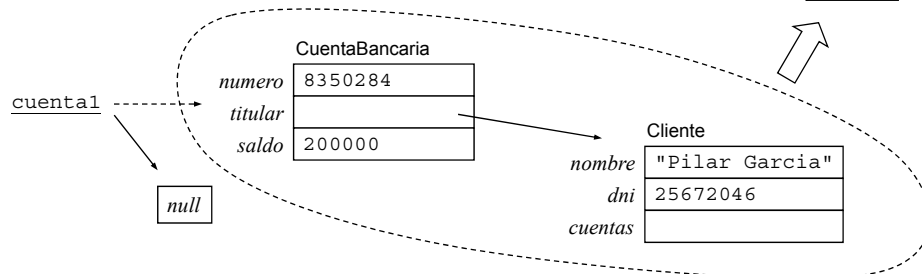
```
CuentaBancaria cuenta1 =  
    new CuentaBancaria (8350284,  
        new Cliente ("Luis Perez", 15165442),  
        200000);  
cuenta1.titular = new Cliente ("Pilar Gomez", 25672046);
```



Destrucción de objetos (II)

- Soltar expresamente una referencia:

```
cuenta1 = null;
```



- Un objeto se libera sólo cuando se sueltan todas las referencias a él
CuentaBancaria cuenta2 = cuenta1;
cuenta1 = null; // El objeto no se libera aún
- Es posible solicitar garbage collection: `System.gc ();`

25

Control de acceso Miembros públicos vs. privados

```
class A {  
    public int x;  
    private int y;  
    public void f () { ... }  
    private void g () { ... }  
    void h () {  
        x = 2;  
        y = 6;  
        f ();  
        g ();  
        A a = new A ();  
        a.x = 2;  
        a.y = 6;  
        a.f ();  
        a.g ();  
    }  
}  
  
class B {  
    void h () {  
        A a = new A ();  
        a.x = 2;  
        a.y = 6; // Error  
        a.f ();  
        a.g (); // Error  
    }  
}
```

26

Control de acceso: otras modalidades

Modalidades de ocultación de variables, métodos y constructores de una clase

| | <u>Clase</u> | <u>Package</u> | <u>Subclase</u> | <u>Cualquiera</u> |
|------------------------------|--------------|----------------|-----------------|-------------------|
| <i>private</i> | X | | | |
| (por defecto) <i>package</i> | X | X | | |
| <i>protected</i> | X | X | X | |
| <i>public</i> | X | X | X | X |

Modalidades de ocultación de clases: *public* ó *package*

27

Packages

- Conjunto de clases definidas en un directorio
- Evitar conflictos de símbolos
- Cada clase pertenece a un package
- Si no se define ningún package para una clase, java la incluye en el package `DefaultPackage`

28

Definir packages

graphics\Circle.java

```
package graphics;  
  
public class Circle {  
    public void paint () {  
        ...  
    }  
    ...  
}
```

graphics\Rectangle.java

```
package graphics;  
  
public class Rectangle {  
    public void paint () {  
        ...  
    }  
    ...  
}
```

29

Utilizar clases de otro package

```
...  
graphics.Circle c = new graphics.Circle ();  
c.paint ();  
...
```

```
import graphics.Circle;  
...  
Circle c = new Circle ();  
c.paint ();  
...
```

Importar clase

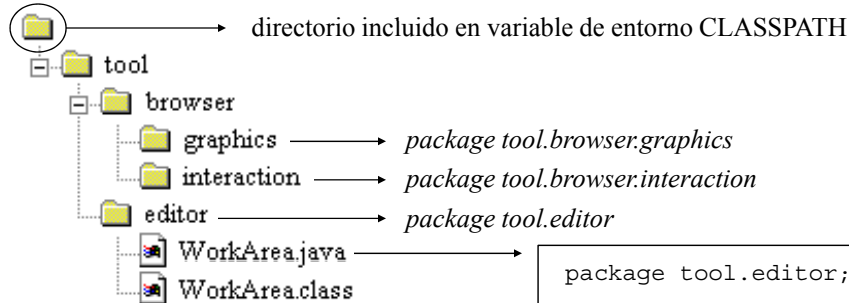
```
import graphics.*;  
...  
Circle c = new Circle ();  
Rectangle r = new Rectangle ();  
c.paint (); r.paint ();  
...
```

Importar todas las clases del package

30

Packages y directorios

- Nombre del package → estructura de directorios
- CLASSPATH: lista de directorios donde java busca packages



- Packages importados automáticamente:
 - java.lang
 - DefaultPackage
 - Package actual

```
package tool.editor;  
public class WorkArea {  
    ...  
}
```

31

Packages predefinidos

| | |
|-----------------------|--------------------------|
| java.applet | java.rmi |
| java.awt | java.rmi.dgc |
| java.awt.datatransfer | java.rmi.registry |
| java.awt.event | java.rmi.server |
| java.awt.image | java.security |
| java.beans | java.security.acl |
| java.io | java.security.interfaces |
| java.lang | java.sql |
| java.lang.reflect | java.text |
| java.math | java.util |
| java.net | java.util.zip |

... (actualmente cerca de 200)

32

Ocultación de variables y métodos dentro de clases

```

class A {
    int w; // package
    private int x;
    protected int y;
    public int z;
    private void f () { ... }
    void h () {
        w = 2;
        x = 6;
        y = 8;
        z = 3;
        f ();
    }
}

class B {
    void h () {
        A a = new A ();
        a.w = 2;
        a.x = 6; // Error
        a.y = 8;
        a.z = 3;
        a.f (); // Error
    }
}
    
```

33

Ocultación de clases dentro de packages

A.java

```

package p1;

// Clase pública
public class A {
    ...
}

// Clase del package
class B {
    ...
}
    
```

```

package p2;

class C {
    void f () {
        p1.A a;
        p1.B b; // Error
    }
}
    
```

34

Ocultación dentro de clases y packages

```
package p1;

public class A {
    int w; // package
    private int x;
    protected int y;
    public int z;
}
```

```
package p2;

class C {
    void h () {
        p1.A a = new p1.A ();
        a.w = 2; // Error
        a.x = 6; // Error
        a.y = 8; // Error
        a.z = 3;
    }
}

class D extends p1.A {
    void h () {
        p1.A a = new p1.A ();
        w = 2; a.w = 2; // Error
        x = 2; a.x = 6; // Error
        z = 3; a.z = 3;
        a.y = 8; // Error
        y = 8;
        D d = new D ();
        d.y = 8;
    }
}
```

Variables estáticas (I)

```
class CuentaCorriente {
    static double interes;
}
```

- Similares a variables globales
- Variables de clase (i.e. estáticas) vs. variables de instancia (por defecto)
- Las variables estáticas pertenecen a la clase, no a los objetos
- Acceso a variables estáticas: desde la clase o desde los objetos

```
CuentaCorriente cuenta = new CuentaCorriente ();
cuenta.interes = 0.3;
CuentaCorriente.interes = 0.2;
```

VARIABLES ESTÁTICAS (II)

- Las variables estáticas son compartidas por todos los objetos de la clase (no se crea una copia por objeto como con las variables de instancia)

```
CuentaCorriente cuenta1 = new CuentaCorriente ();  
CuentaCorriente cuenta2 = new CuentaCorriente ();  
cuenta1.interes = 0.3;  
cuenta2.interes = 0.4;  
System.out.println (cuenta1.interes); —————> 0.4
```

- La memoria de las variables estáticas se reserva al cargar la clase en el intérprete Java

37

MÉTODOS ESTÁTICOS

```
class CuentaCorriente {  
    long numero;  
    static long ncuentas;  
    static long generarNumero () {  
        return ncuentas++;  
    }  
}
```

- Similares a funciones globales
- Pueden ser invocados desde la clase o desde los objetos

```
CuentaCorriente cuenta = new CuentaCorriente ();  
cuenta.numero = cuenta.generarNumero ();  
cuenta.numero = CuentaCorriente.generarNumero ();
```

38

Métodos estáticos: restricciones

- Pueden acceder a variables y métodos estáticos de la misma clase
- No pueden acceder a `this`
- No pueden acceder a variables ni métodos de instancia de la clase

```
static long generarNumero () {  
    numero = numcuentas++; // Error  
    return ncuentas;  
}
```
- Los métodos de instancia sí pueden acceder a variables y métodos estáticos

```
void cobrarIntereses () {  
    saldo += saldo * interes / 100;  
}  
CuentaBancaria () { // Constructor  
    numero = generarNumero ();  
}
```
- Un constructor no puede ser estático

39

Variables y métodos estáticos típicos

- Variables estáticas: `System.in`, `System.out`, `Math.PI`,
`Integer.MAX_VALUE`, etc.
- Métodos estáticos: `System.exit(int)`, `Math.sqrt(double)`,
`Math.cos(double)`, `Integer.valueOf(String)`, `main`, etc.

40

Bloques de inicialización estática

```
class A {  
    static double num;  
    String x;  
    static {  
        num = 5.2;  
        x = "Hola"; // Error  
    }  
}
```

```
class A {  
    static double num = 5.2;  
    String x = "Hola";  
}
```

- Se ejecutan sólo una vez: al cargar la clase
- Se permite un número arbitrario de bloques en una clase
- No pueden referenciar variables de instancia ni métodos de instancia
- Útiles para inicializar variables estáticas con tratamiento de errores
- Cuando es posible, mejor inicializar en la declaración o en constructores

41

Variables final

- Similar a constantes, no se puede modificar su valor, inicialización obligatoria

```
class Circle {  
    final double PI = 3.141592653589793;  
    void f () {  
        PI = 3.0; // Error  
    }  
}
```

- Más eficiente: `static final`
- Un constructor no puede ser `final`

42

Librerías de clases fundamentales

Librería estándar Java Algunas clases básicas

- Manejo de strings
- Entrada / salida
 - Estándar
 - Disco
- Funciones matemáticas
- Tipos primitivos
- Listas
- Otros

Cadenas de caracteres: La clase java.lang.String

- String encapsula cadenas de caracteres y su manipulación
- String \neq char[]
- Los caracteres de un String no se pueden modificar
- Constructores

```
String (char[])  
String (String)
```

- Creación de strings

```
char[] chars = {'a', 'b', 'c'};  
String s = new String (chars);  
String s1 = "Hello";  
String s2 = new String (s1);
```

- Literales: Java crea objetos de tipo String para los literales

45

Métodos de la clase String (I)

- Acceso (la notación str[n] no existe para String)

```
String str = "abcdabc";  
str.length (); // → 7  
str.charAt (4); // → 'a'  
str.substring (3, 5); // → "da"  
str.indexOf ("bc"); // → 1  
str.lastIndexOf ("bc"); // → 5
```

- Manipulación (se devuelve un String)

```
str.concat ("xyz"); // → "abcdabcxyz"  
str.toUpperCase (); // → "ABCDABC"  
str.replace ('b', 'x'); // → "axcdaxc"  
" Hola que tal ".trim (); // → "Hola que tal"
```

46

Métodos de la clase String (II)

- Conversión a String: `static String valueOf (<cualquier tipo>)`

```
String.valueOf (2.34567);           // → "2.34567"  
String.valueOf (34);                // → "34"  
String.valueOf (new Plano3D ());    // → "Plano3D@1d07be"
```

- Comparación

```
String s1 = "abc", s2 = new String ("abc"), s3 = "abx";  
s1 == s2;                // → false  
s1.equals (s2);         // → true  
s1.compareTo (s3);     // → -21 < 0
```

47

Cadenas de caracteres: La clase `java.lang.StringBuilder`

- Desde Java 1.5, recomendado en lugar de `StringBuffer` (equivalente)
- Los caracteres de un `StringBuilder` sí se pueden modificar
- Los objetos de tipo `StringBuilder` gestionan automáticamente su capacidad
 - Toman una capacidad inicial
 - La incrementan cuando es necesario
- Constructores

```
StringBuilder()           // Capacidad inicial: 16 caracteres  
StringBuilder(int c)     // Capacidad inicial: c caracteres  
StringBuilder(String s)  // Capacidad inicial:  
                        // s.length() + 16 caracteres
```

48

Métodos de StringBuilder (I)

- Acceso (igual que para String): `length()`, `charAt(int)`, ...
- Conversión a String: `toString()`
- Modificación de la cadena

```
StringBuilder str = new StringBuilder ("abcdef");  
str.setCharAt (2, 'q'); // str = "abqdef"  
str.append ("ghi"); // str = "abqdefghi"  
str.insert (3, "xyz"); // str = "abqxyzdefghi"  
str.insert (6, 1.23); // str = "abqxyz1.23defghi"  
str.delete (2, 10); // str = "abdefghi" (versión 1.2)  
str.reverse (); // str = "ihgfedba"
```

49

Métodos de StringBuilder (II)

- Manipulación de longitud y capacidad
`length()`, `capacity()`
`setLength(int)`, `ensureCapacity(int)`
- Operador de concatenación: internamente se utiliza un `StringBuilder`

```
{ "DNI de " + cliente.nombre + ": " + cliente.dni  
new StringBuilder().append("DNI de ")  
    .append(cliente.nombre).append(": ")  
    .append(cliente.dni).toString()
```

50

Entrada y salida estándar: La clase java.lang.System

System.out, System.err

- Objeto de la clase `java.io.PrintStream`
- Métodos: `print(<cualquier tipo>)`, `println(<cualquier tipo>)`, `flush()`, `format(<String>, <cualquier tipo>, ...)`, `printf(...)`
- `format` y `printf` emiten `java.util.IllegalFormatException`

System.in

- Objeto de la clase `java.io.InputStream`
- Métodos: `read()`, `read(byte[])`, `read(byte[], int, int)`
- Emiten `java.io.IOException`

Redireccionamiento de la E/S estándar:

- `System.setOut(PrintStream)`, `System.setErr(PrintStream)`
- `System.setIn(InputStream)`

51

Entrada estándar con java.io.BufferedReader

`InputStream` → `InputStreamReader` → `BufferedReader` → `String`
(`System.in`)

```
import java.io.*;

public static void main (String args[]) throws IOException {
    BufferedReader reader =
        new BufferedReader (new InputStreamReader (System.in));
    String str = reader.readLine ();
}
```

52

Alternativa: E/S estándar con java.io.Console

- Se obtiene con `System.console()`
 - Métodos como `readLine()`, `format(...)`, `printf(...)`, `readPassword()`
- ```
public static void main (String args[]) throws IOException {
 Console c = System.console ();
 String str = c.readLine ("Introduzca su nombre: ");
 ...
}
```
- Excepciones `java.io.IOException`, `java.util.IllegalFormatException`

53

## Escritura en fichero en modo texto con java.io.PrintStream

- Permite escribir en modo texto (como con `System.out`)
- ```
PrintStream printer =  
    new PrintStream (new FileOutputStream ("abc.txt"));  
printer.print ("Dos + " + 2);  
printer.println (" = " + (2+2));  
...  
printer.close ();
```
- Emite `java.io.IOException`

54

Lectura de fichero en modo texto con `java.io.BufferedReader`

- Permite leer en modo texto (como con `System.in`)

```
BufferedReader reader =  
    new BufferedReader (new FileReader ("abc.txt"));  
String str = reader.readLine ();  
...  
reader.close ();
```

- Emite `java.io.IOException`

55

Entrada y salida en fichero binario: La clase `java.io.RandomAccessFile`

- Permite leer y escribir tipos primitivos (no objetos) en forma binaria
- Abrir un fichero

```
// Modalidades: "r", "rw"  
RandomAccessFile f = new RandomAccessFile ("abc.txt","r");  
f.close ();
```

- Métodos:

```
readInt () → int, readBoolean () → boolean, etc.  
writeInt (int), writeBoolean (boolean), etc.  
getFilePointer (), seek (long), length (), setLength ()
```

- Emiten `java.io.IOException`

56

Funciones matemáticas: La clase java.lang.Math

- **Constantes:** `Math.PI`, `Math.E`
- **Métodos:** `sqrt(double)`, `pow(double, double)`, `random()`,
`abs(double)`, `max(double, double)`, `round(double)`,
`cos(double)`, `sin(double)`, `tan(double)`, `acos(double)`,
`exp(double)`, `log(double)`, etc.

(Existen versiones `float`, `long`, `int` para `abs`, `max`, `min`, `round`)

57

Clases para tipos numéricos (I) (package java.lang)

- `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`
- Encapsulamiento (wrapping) de valores para manejo genérico de tipos
`Integer n = new Integer (54); // n es un objeto`
`// 54 es un dato primitivo`
- Conversión automática tipo primitivo ↔ objeto por el compilador
`n = 27; // n es un objeto, 27 es un dato primitivo`
`int i = n; // i es un dato primitivo, n es un objeto`
- Conversión a `String` y viceversa
`n = Integer.valueOf ("91"); // String → Integer`
`String s1 = m.toString (); // Integer → String`
`String s2 = String.valueOf (27.5); // int → String`
`// etc.`

58

Clases para tipos numéricos (II)

- Constantes (variables de clase)

```
Integer.MIN_VALUE → -2147483648
Integer.MAX_VALUE → 2147483647
Float.MIN_VALUE   → 1.4E-45
Float.MAX_VALUE   → 3.4028235E38
// etc.
Float.NEGATIVE_INFINITY
Float.POSITIVE_INFINITY
Float.NaN
```

} public
static
final

59

Clases para otros tipos primitivos

- Clase `java.lang.Character`

```
Character c = 'a';           // char → Character
char ch = c;                 // Character → char
Character.isDigit ('2');     // → true
Character.isLetter ('a');    // → true
Character.isLowerCase ('a'); // → true
ch = Character.toUpperCase ('a'); // ch = 'A'
```

- Clase `java.lang.Boolean`

```
Boolean b1 = true;           // boolean → Boolean
boolean b = b1;              // Boolean → boolean
Boolean b2 = Boolean.valueOf ("true"); // String → Boolean
String s1 = b1.toString ();   // Boolean → String
String s1 = String.valueOf (true); // boolean → String
```

60

La clase java.util.ArrayList

- Lista de longitud variable de cualquier tipo de objetos
- Similar a array, pero su capacidad aumenta o disminuye dinámicamente
- Longitud, capacidad, incremento
- Constructores

```
ArrayList lista = new ArrayList (); // Capacidad inicial: 10  
ArrayList lista = new ArrayList (100); // Capacidad inicial: 100
```

- Ajustar la capacidad: `ensureCapacity(int)`

61

Métodos de ArrayList (I)

```
ArrayList lista = new ArrayList (); // lista = { }
```

- Inserción

```
lista.add (2); // Warning (antes error)  
lista.add (new Integer (2)); // lista = { 2 }  
lista.add (new Float (4.5)); // lista = { 2, 4.5 }  
lista.add (1, "Hola"); // lista = { 2, "Hola", 4.5 }  
lista.set (0, new Character ('b')); // lista = { 'b', "Hola", 4.5 }
```

- Eliminar elementos:

```
lista.remove (new Float (4.5)); // lista = { 'b', "Hola" }  
lista.remove (0); // lista = { "Hola" }
```

62

Métodos de ArrayList (II)

- Acceso

```
// Suponiendo que lista = { 'b', "Hola", 4.5 }  
lista.get (1);           // → "Hola"  
lista.indexOf ("Hola"); // → 1 (-1 si no encontrado)
```

- Longitud y capacidad: `size()`, `isEmpty()`, `ensureCapacity(int)`, `trimToSize()`
- Conversión a array: `toArray()`
- ArrayList tipado: `ArrayList<Double> lista = new ArrayList<Double> ();`
- Iteración: `iterator() → Iterator`

63

Iteración sobre un ArrayList

- Métodos de la interfaz `Iterator`

| | |
|---|--|
| } | <code>hasNext() → boolean</code> |
| | <code>next() → Object (o tipo especificado)</code> |

- Ejemplo (suponiendo que la clase `Punto2D` tiene un método `print()`)

```
ArrayList<Punto2D> lista = new ArrayList<Punto2D> ();  
lista.add (new Punto2D (0, 1));  
lista.add (new Punto2D (2, -3));  
lista.add (new Punto2D (-1, 1));  
System.out.println ("Los elementos de la lista son: ");
```

```
Iterator<Punto2D> iter = lista.iterator ();  
while (iter.hasNext()) iter.next () .print ();
```

 } Con iterador

```
for (int i = 0; i < lista.size (); i++)  
    lista.get (i) .print ();
```

 } Sin iterador

```
for (Punto2D p : lista) p.print ();
```

 } Enhanced for

64

Otras clases

- Package `java.util`: `LinkedList`, `HashMap`, `Set`, `Collections`, `Date`, `StringTokenizer`...
- Package `java.text`: `DateFormat`, `DecimalFormat`
- Package `java.math`: `BigDecimal`, `BigInteger` (precisión y capacidad arbitrarias)
- La clase `java.lang.System`: `in`, `out`, `exit(int)`
- La clase `java.lang.Runtime`: `getRuntime()`, `exec(String)`, `exit(int)`

65

Enumeraciones (I)

- Enumeración simple
 - Definición:

```
public enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }
```
 - Utilización:

```
Day d = Day.SATURDAY;
```
- Una enumeración es una subclase de `java.lang.Enum`
 - El método `Enum.values()` devuelve un array con los valores de la enumeración
- Clases de enumeración: igual que una clase pero con una lista fija de instancias

```
public enum Planet {  
    MERCURY (3.303e+23, 2.4397e6), VENUS (4.869e+24, 6.0518e6),  
    EARTH (5.976e+24, 6.37814e6), MARS (6.421e+23, 3.3972e6),  
    JUPITER (1.9e+27, 7.1492e7), SATURN (5.688e+26, 6.0268e7),  
    URANUS (8.686e+25, 2.5559e7), NEPTUNE (1.024e+26, 2.4746e7);  
    private final double mass, radius;  
    Planet (double m, double r) { mass = m; radius = r; }  
    public static final double G = 6.67300E-11; // gravit. constant  
    double gravity () { return G * mass / (radius * radius); }  
    double weight (double m) { return m * gravity(); }  
    ...  
}
```

66

Enumeraciones (II)

```
...
public static void main (String[] args) {
    double earthWeight = Double.valueOf (args[0]);
    double mass = earthWeight/EARTH.gravity ();
    for (Planet p : Planet.values ())
        System.out.println ("Your weight on " + p +
                               " is " + p.weight (mass));
    }
}
```

67

Tipos genéricos

- Permiten definir clases parametrizadas por un tipo
- El compilador se encarga de hacer los castings
- Los errores se detectan en tiempo de compilación, no de ejecución
- Tipos restringidos, wildcards, métodos genéricos

68

Ejemplo (I)

```

abstract class Figura {
    abstract double area ();
}

class Circulo extends Figura {
    Punto2D centro;
    double radio;
    Circulo (Punto2D p, double r) { centro = p; radio = r; }
    double area () { return Math.PI * radio * radio; }
}

class Rectangulo extends Figura {
    double left, top, width, height;
    Rectangulo (double l, double t, double w, double h) {
        left = l; top = t; width = w; height = h;
    }
    double area () { return width * height; }
}

```

69

Ejemplo (II)

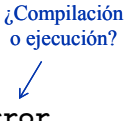
```

class Box<T> {
    T t;
    void set (T elem) { t = elem; }
    T get () { return t; }
}

// Bloque main
...
Box<Circulo> bCirc = new Box<Circulo> ();
Box<Rectangulo> bRect = new Box<Rectangulo> ();
bCirc.set (new Circulo (new Punto2D (2, 1), 3));
Circulo c = bCirc.get (); // no cast!
bRect.set (new Rectangulo (2, 1, 5, 4));
bCirc.set (new Rectangulo (2, 1, 5, 4)); // Error
...

```

¿Compilación o ejecución?



70

Tipo restringido

```
class Box<T extends Figura> {  
    T t;  
    void set (T elem) { t = elem; }  
    T get () { return t; }  
    double area () { return t.area (); }  
}  
  
// Bloque main  
...  
Box<Circulo> bCirc = new Box<Circulo> ();  
Box<Rectangulo> bRect = new Box<Rectangulo> ();  
Box<Punto2D> bPunto = new Box<Punto2D> (); // Error  
...
```

¿Compilación o ejecución?
↓

71

Jerarquías de tipos genéricos

- Box<Circulo> no es subtipo de Box<Figura> ← ¿Por qué?
- Box<Figura> b = new Box<Circulo> (); // **Error**
- ArrayList<Object> a = new ArrayList<String> (); // **Error**
- Box<Circulo> sí es subtipo de Box<? extends Figura>
- Box<? extends Figura> b = new Box<Circulo> ();
- b.set (new Rectangulo (2, 1, 5, 4)); // **Error**
- ArrayList<? extends Object> a1 = new ArrayList<String> ();
- ArrayList<?> a2 = new ArrayList<String> ();
- Box<Figura> es subtipo de Box<? super Circulo>
- Box<? super Circulo> b1 = new Box<Figura> ();
- Box<? super Circulo> b2 = new Box<Rectangulo> (); // **Error**
- Restricciones de los tipos wildcard
- b.set (new Circulo (...)); // **Error**
- a1.set (new Object ()); // **Error**
- a2.set (new Object ()); // **Error**

72

Otras limitaciones

No es legal...

```
new T()
```

```
new T[]
```

```
(T) <expresión>
```

Debido al “type erasure” por compatibilidad hacia atrás