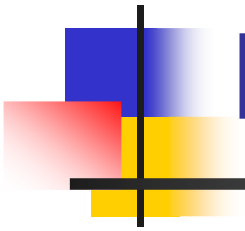


Programación Java

Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

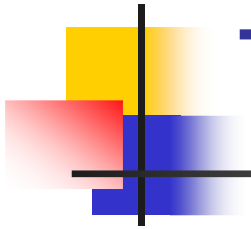
- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

Desarrollo de interfaces gráficas de usuario

- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

Programación Java avanzada

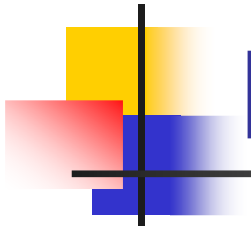
- Hilos
- Funciones de red



Tecnología Java

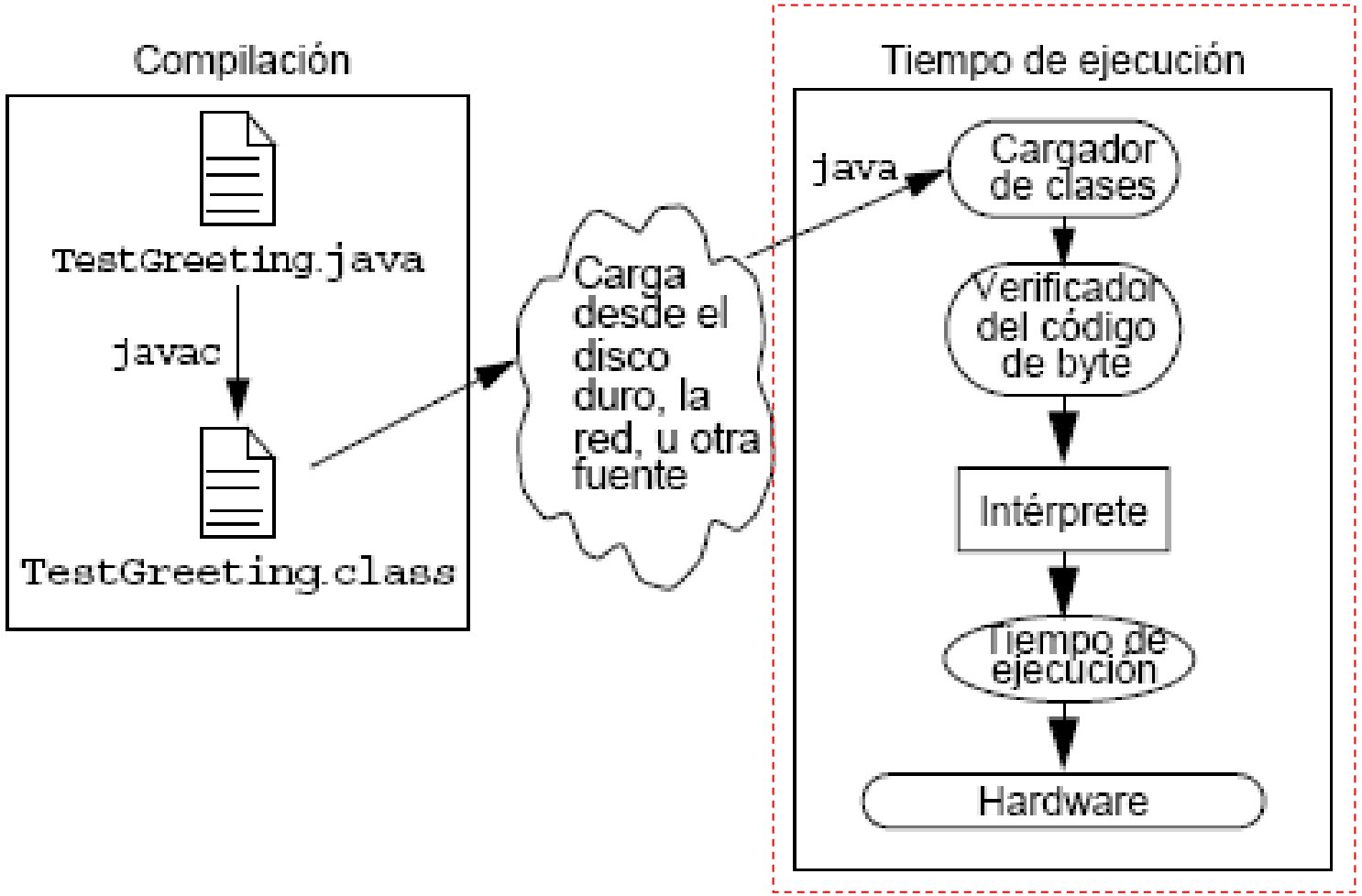
- Un lenguaje de programación
- Un entorno de desarrollo
  - El JDK
- Un entorno de aplicaciones
  - El JRE → la JVM
- Un entorno de implantación
  - En local la JVM, en remoto el Navegador

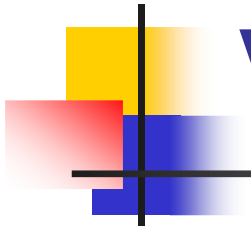
Características de la plataforma



- La JVM
  - Virtual
  - Portable entre S.O.
- El reciclaje de memoria
- El JRE
- JVM Tool Interface: interfaz de herramientas de JVM

Entorno de ejecución de Java (JRE)

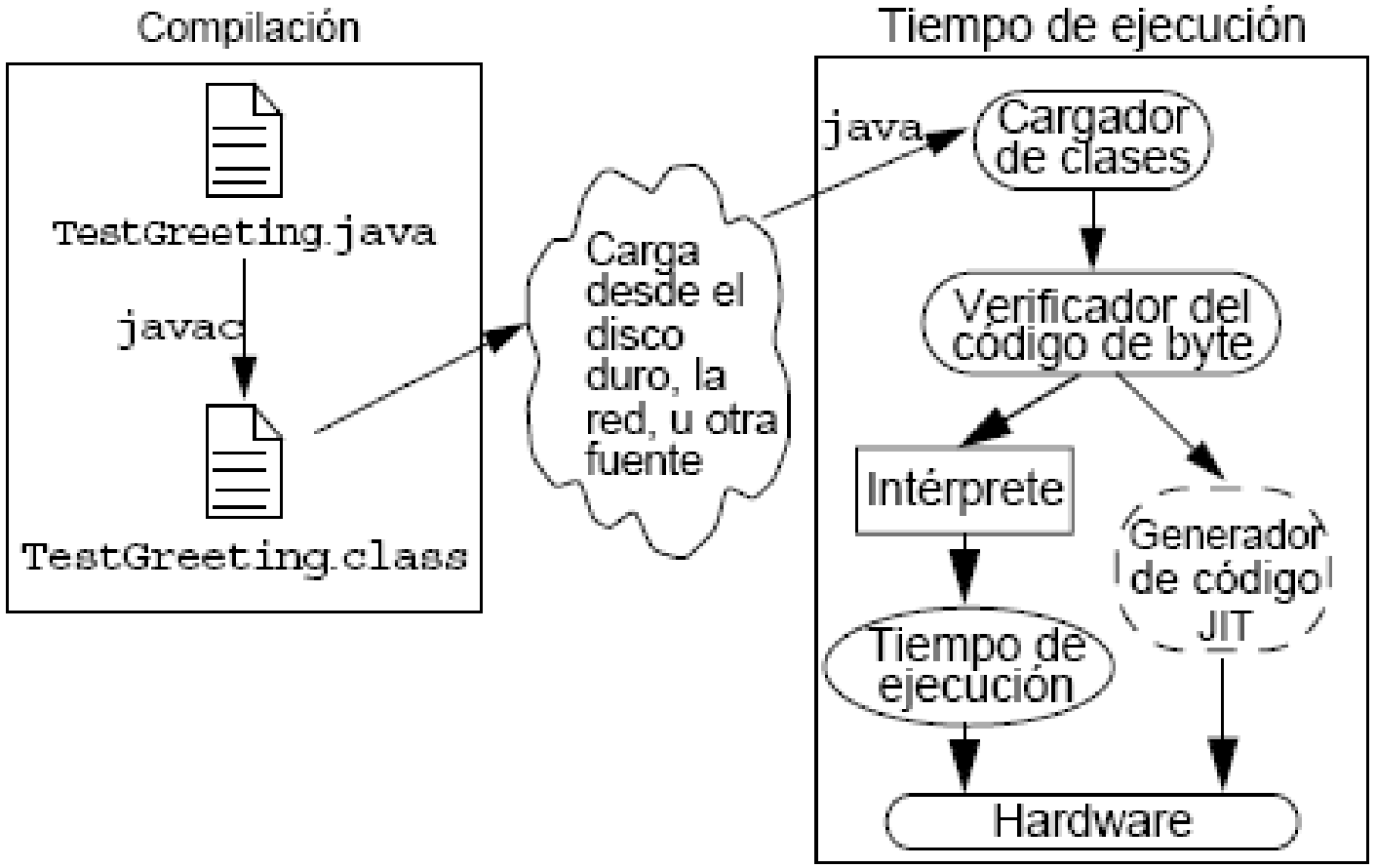




Verificación de byte codes

- Las clases son conformes con el formato de archivos de clases de la especificación de JVM
- No se han producido intentos de infringir las reglas de acceso
- El código no provoca desbordamiento ni falta de operandos en la pila
- Todos los tipos de parámetros de los códigos operativos son correctos
- No se han realizado conversiones de datos irregulares tales como la conversión de enteros en referencias a objetos

JRE con compilación JIT



Ejemplo

Código 1-1 AplicaciónTestGreeting.java

```
1   //
2   // Ejemplo de aplicación de "Saludo"
3   //
4   public class TestGreeting {
5      public static void main (String[] args) {
6         Greeting hello = new Greeting();
7         hello.greet();
8      }
9   }
```

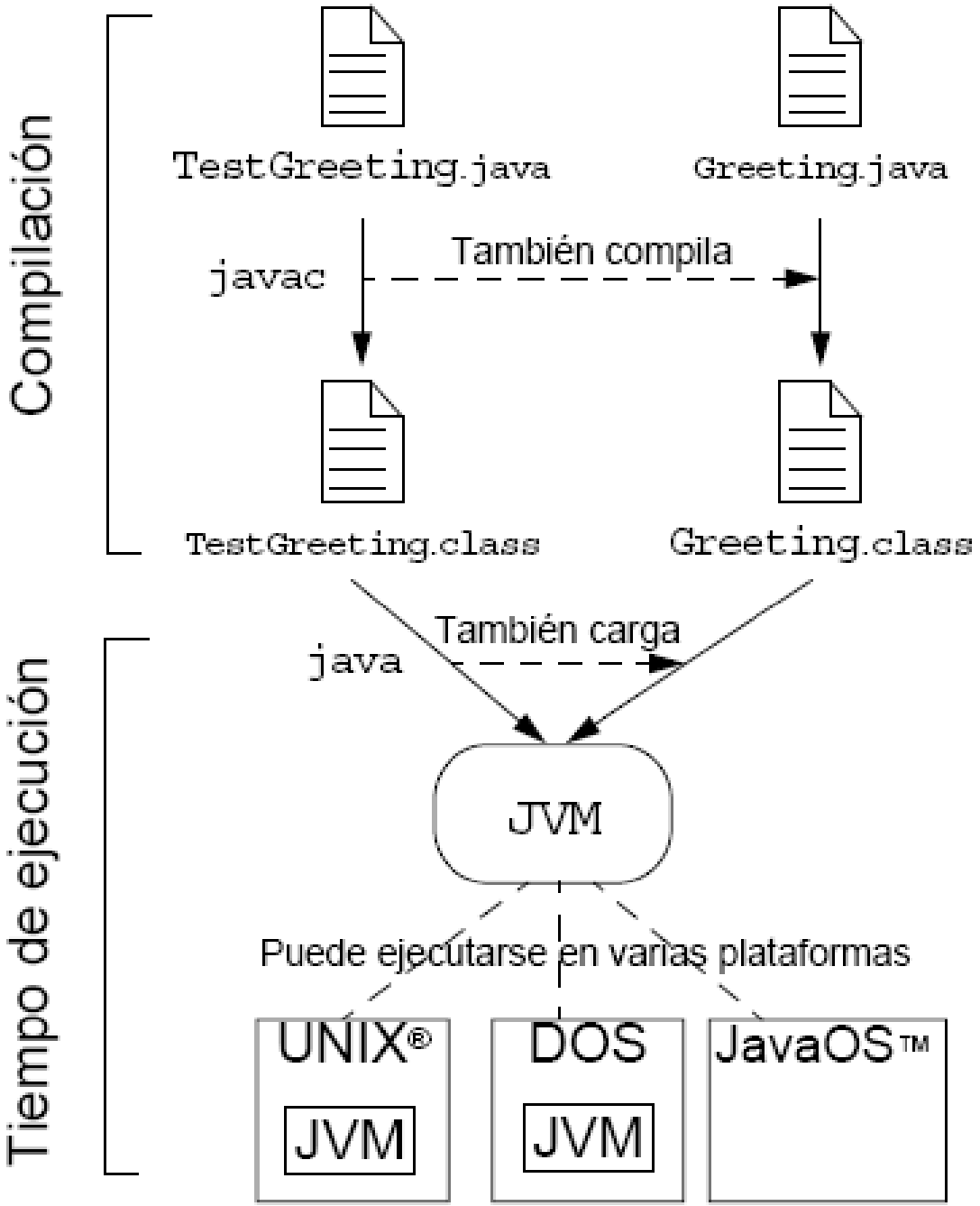
Código 1-2 Clase Greeting.java

```
1   public class Greeting {
2       public void greet() {
3          System.out.println("hola");
4       }
5   }
```

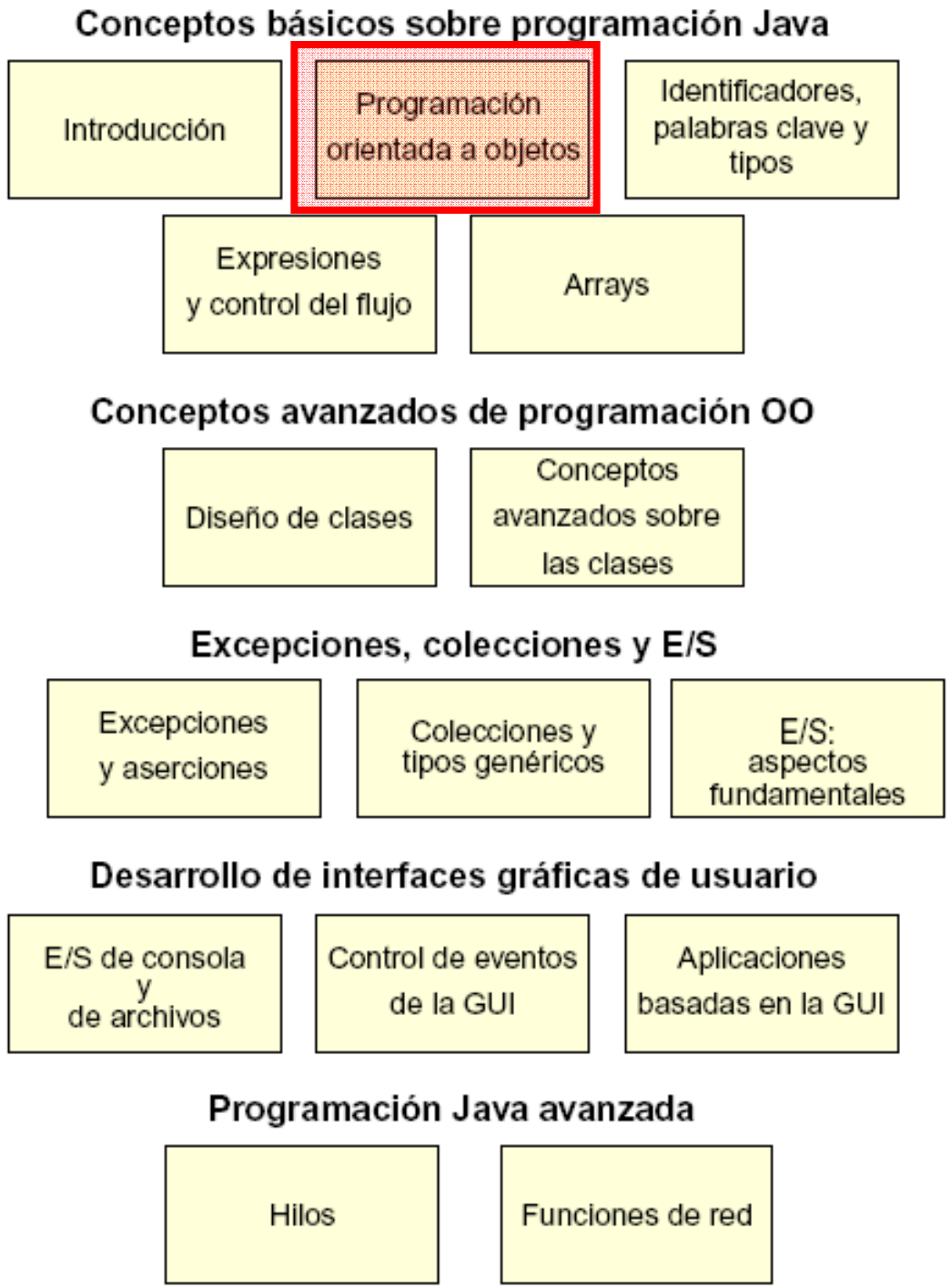
```
javac TestGreeting.java
```

```
java TestGreeting
```

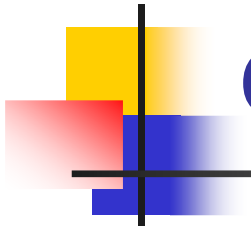
Compilación y ejecución



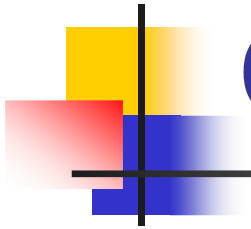
Contenidos


Conceptos básicos sobre programación Java
Introducción
Programación orientada a objetos
Identificadores, palabras clave y tipos
Expresiones y control del flujo
Arrays
Conceptos avanzados de programación OO
Diseño de clases
Conceptos avanzados sobre las clases
Excepciones, colecciones y E/S
Excepciones y aserciones
Colecciones y tipos genéricos
E/S: aspectos fundamentales
Desarrollo de interfaces gráficas de usuario
E/S de consola y de archivos
Control de eventos de la GUI
Aplicaciones basadas en la GUI
Programación Java avanzada
Hilos
Funciones de red

Requisitos funcionales: ejemplo ??



- El software debe aplicarse a una sola compañía de transporte.
- La compañía cuenta con una flota de vehículos que transporta cajas.
- El peso de las cajas es el único factor importante que debe tenerse en cuenta al cargar un vehículo.
- La empresa posee dos tipos de vehículos: camiones y barcazas de transporte fluvial.
- Las cajas se pesan utilizando el kilogramo como unidad de medida, pero los algoritmos para calcular la potencia de motor necesaria deberá contabilizar la carga total del vehículo medida en newtons.
- Deberá utilizar una interfaz gráfica para hacer el seguimiento de las cajas que se van agregando a los vehículos.
- También deberá generar varios informes a partir de los registros de fletes.



Clases en Java (y en OO)

- La clase como prototipo de objetos (factoría)
  - Define atributos (datos)
  - Define métodos (comportamiento)
- Conceptos clave:
  - Ocultación de información
  - Encapsulación
  - Herencia
  - Polimorfismo

Declaraciones:

```
<modificador>* class <nombre_clase> {
   <declaración_atributo>*
   <declaración_constructor>*
   <declaración_método>*
}
```

De clase

```
<modificador>* <tipo> <nombre> [ = <valor_inicial>];
```

De atributo

```
<modificador>* <tipo_retorno> <nombre> ( <argumentos>* ) {
   <sentencia>*
}
```

De método

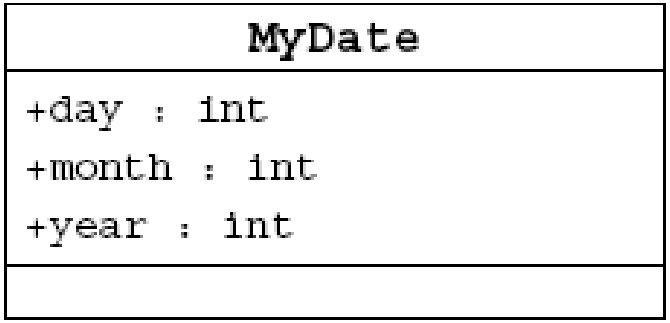
Declaraciones: ejemplos

```
1   public class Ejemplo {
2     private int x;
3     private float y = 10000.0F;
4     private String name = "Hotel Mediodía";
5   }
```

```
1   public class Vehiculo {
2     private double cargaMax;
3     public void setCargaMax(double valor) {
4       cargaMax = valor;
5     }
6   }
```

```
1   public class Perro {
2     private int peso;
3     public int getPeso() {
4       return peso;
5     }
6     public void setPeso(int newPeso) {
7       if ( newPeso > 0 ) {
8         peso = newPeso;
9       }
10    }
11  }
```

Ocultación de información

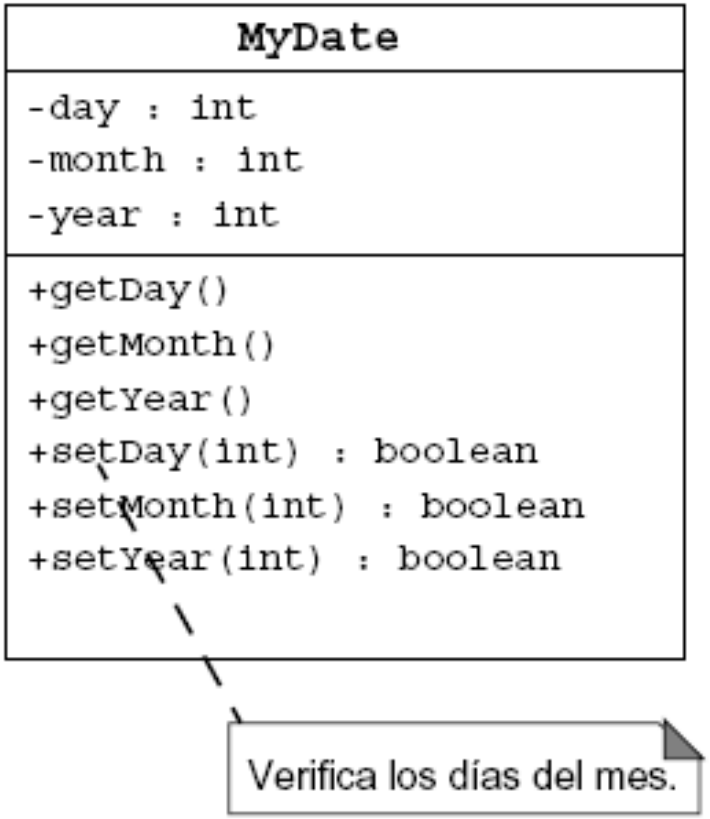


- Impedir manipulaciones descontroladas de los atributos

```
public class MyDate {
   public int day;
   public int month;
   public int year;
}
```

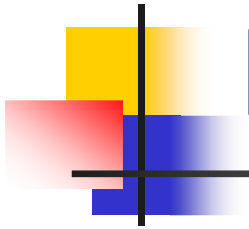
```
d.day = 32;
// día no válido

d.month = 2; d.day = 30;
// posible pero incorrecto

d.day = d.day + 1;
// no hay comprobación de comportamiento cíclico
```

Ocultación: private, protected



- Acceso controlado a los atributos

```
MyDate d = new MyDate();

d.setDay(32);
// día no válido, devuelve false

d.setMonth(2); d.setDay(30);
// posible pero incorrecto, setDay devuelve false

d.setDay(d.getDay() + 1);
```



Encapsulación

MyDate
-date : long
+getDay() : int +getMonth() : int +getYear() : int +setDay(int) : boolean +setMonth(int) : boolean +setYear(int) : boolean -isDayValid(int) : boolean

- Usando los mecanismos de ocultación conseguimos encapsulación
  - Ofrecer una vista pública (interfaz) sin dar detalles de cómo es la implementación

Constructores

```
1    public class Perro {
2      private int peso;
3
4      public Perro() {
5          peso = 42;
6      }
```

```
[<modificador>] <nombre_clase> ( <argumentos>* ) {
  <sentencia>*
}
```

- Sin argumentos
  - Por defecto
- Con argumentos
  - Anula la generación por defecto de constructor sin argumentos
  - A veces se necesitan los dos aunque el constructor sin argumento no haga nada

Organización de las clases en paquetes

[<declaración_paquete>]
*<declaración_importación>**
<declaración_clase>+

```
1   package transporte.informes;
2
3   import transporte.dominio.*;
4   import java.util.List;
5   import java.io.*;
6
7   public class InformeCapacidadVehiculo {
8      private List vehiculos;
9      public void generarInforme(Writer output) {
10       // código para generar el informe
11     }
12  }
```

- Orden importante
- Sólo una clase pública, posible varias private class
- Nombre del archivo fuente igual al de la clase pública

Paquetes: declaración e importación

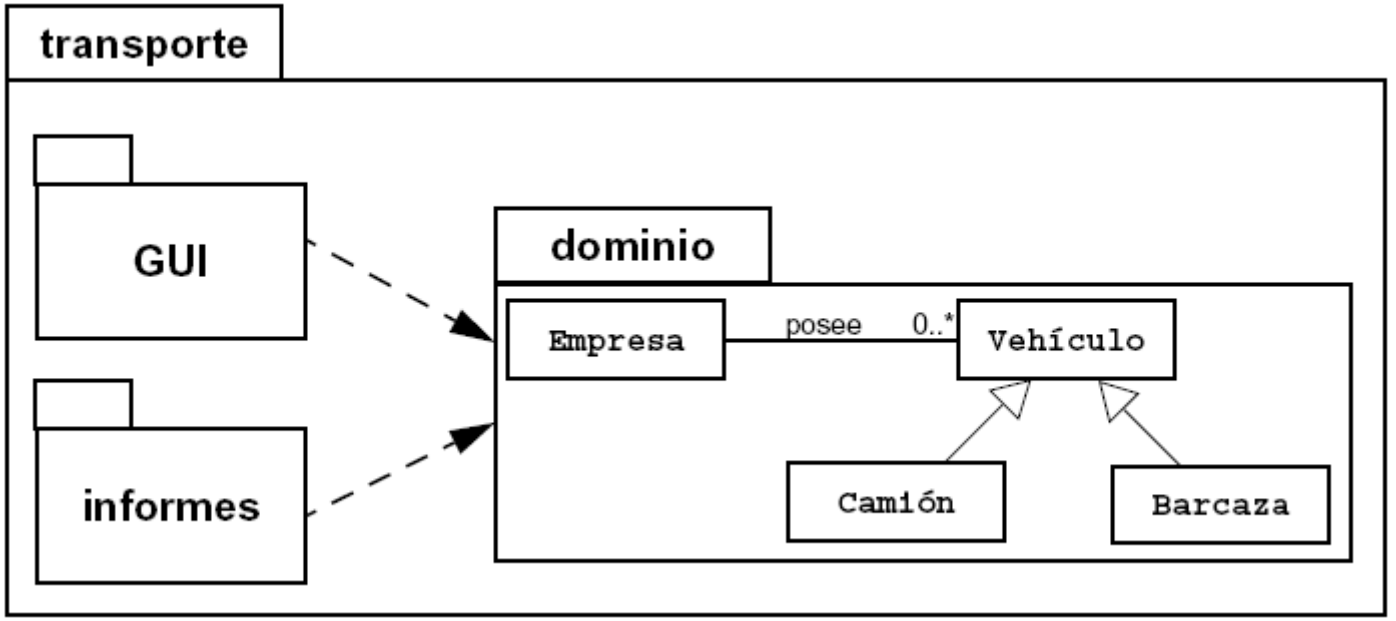
- Declaración de paquete

```
package <nombre_paq_superior>[.<nombre_paq_subordinado>]*;
```

- Declaración de importación

```
import
<nombre_paq>[.<nombre_paq_subordinado>].<nombre_clase>;

import <nombre_paq>[.<nombre_paq_subordinado>].*;
```

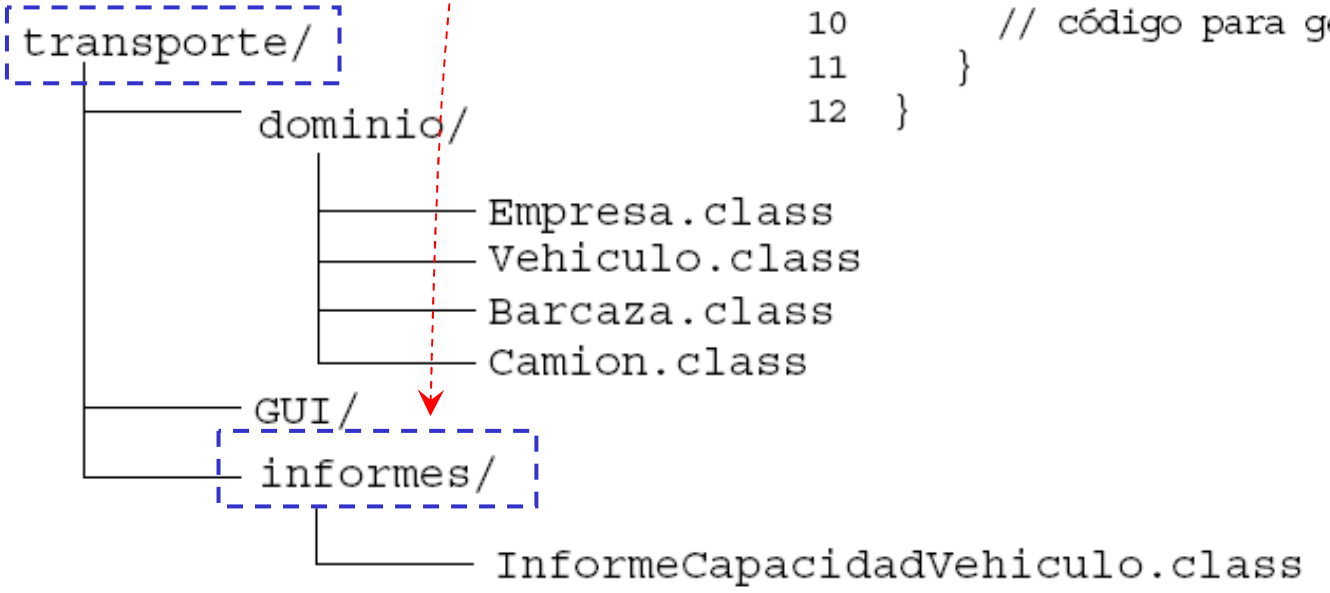
Agrupación en paquetes



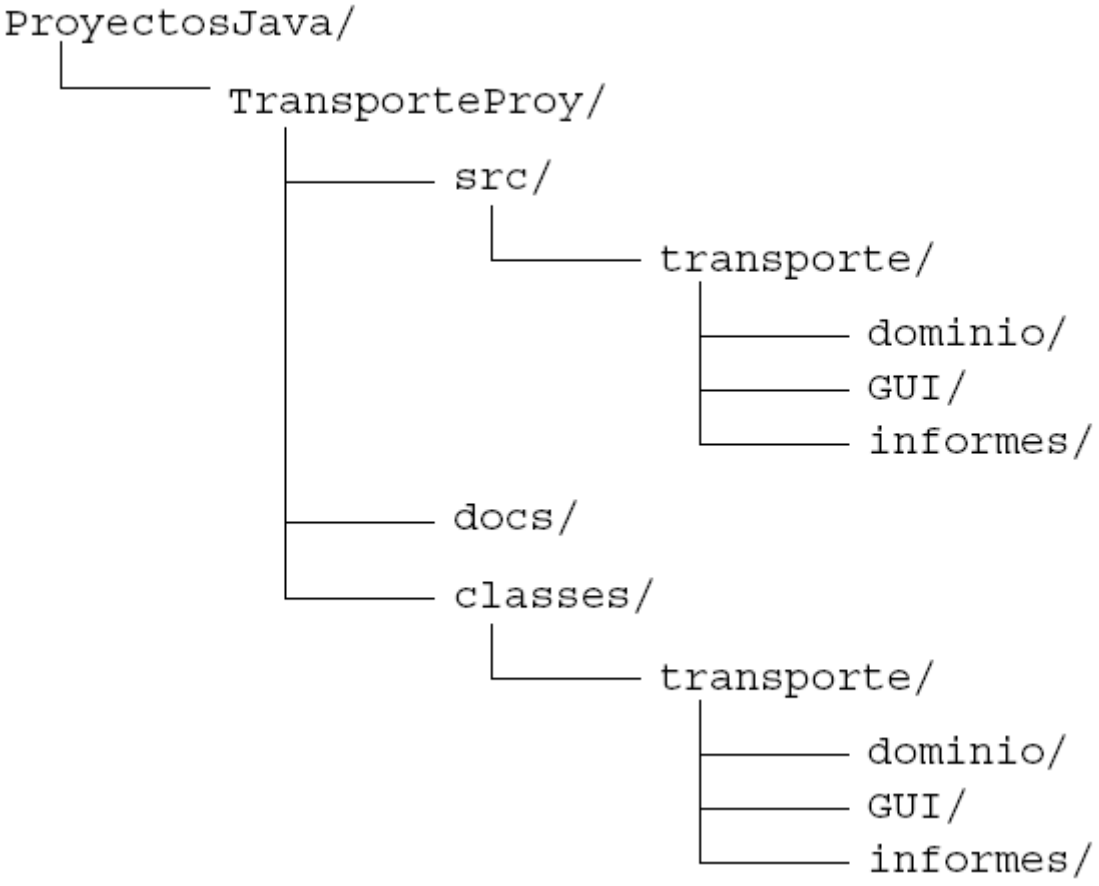
- Si el sistema software es "grande" se subdivide en paquetes
- Cuidado con las dependencias entre paquetes

Traslación de paquete a directorios

```
1   package transporte.informes;
2
3   import transporte.dominio.*;
4   import java.util.List;
5   import java.io.*;
6
7   public class InformeCapacidadVehiculo {
8      private List vehiculos;
9      public void generarInforme(Writer output) {
10        // código para generar el informe
11     }
12  }
```

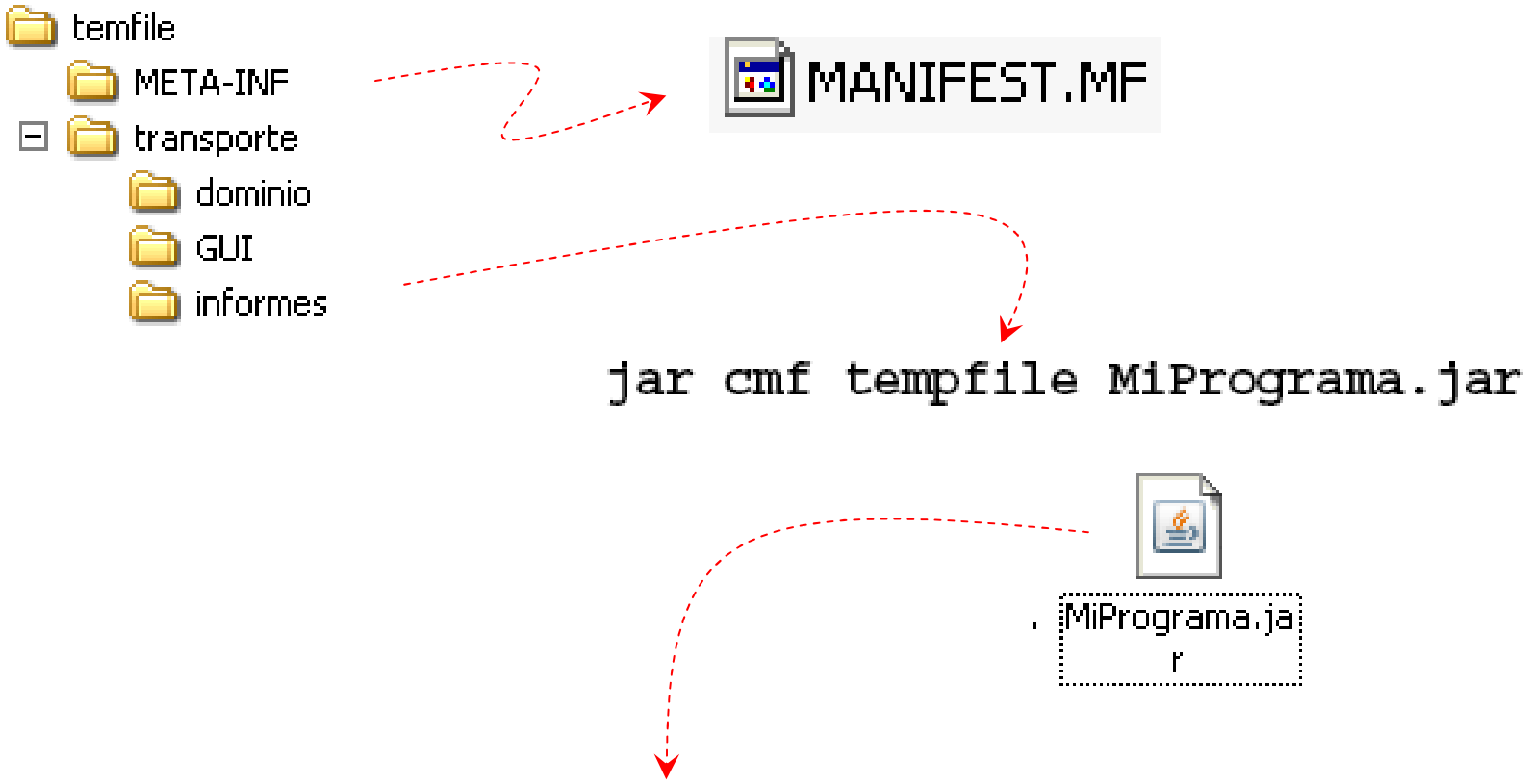


Estructura típica de un proyecto Java




```
cd ProyectosJava/TransporteProy/src
javac -d ../classes transporte/dominio/*.java
```

Empaquetado de clases: jar



Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

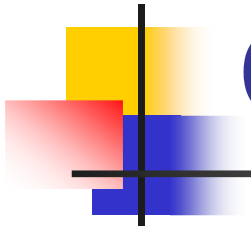
- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

Desarrollo de interfaces gráficas de usuario

- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

Programación Java avanzada

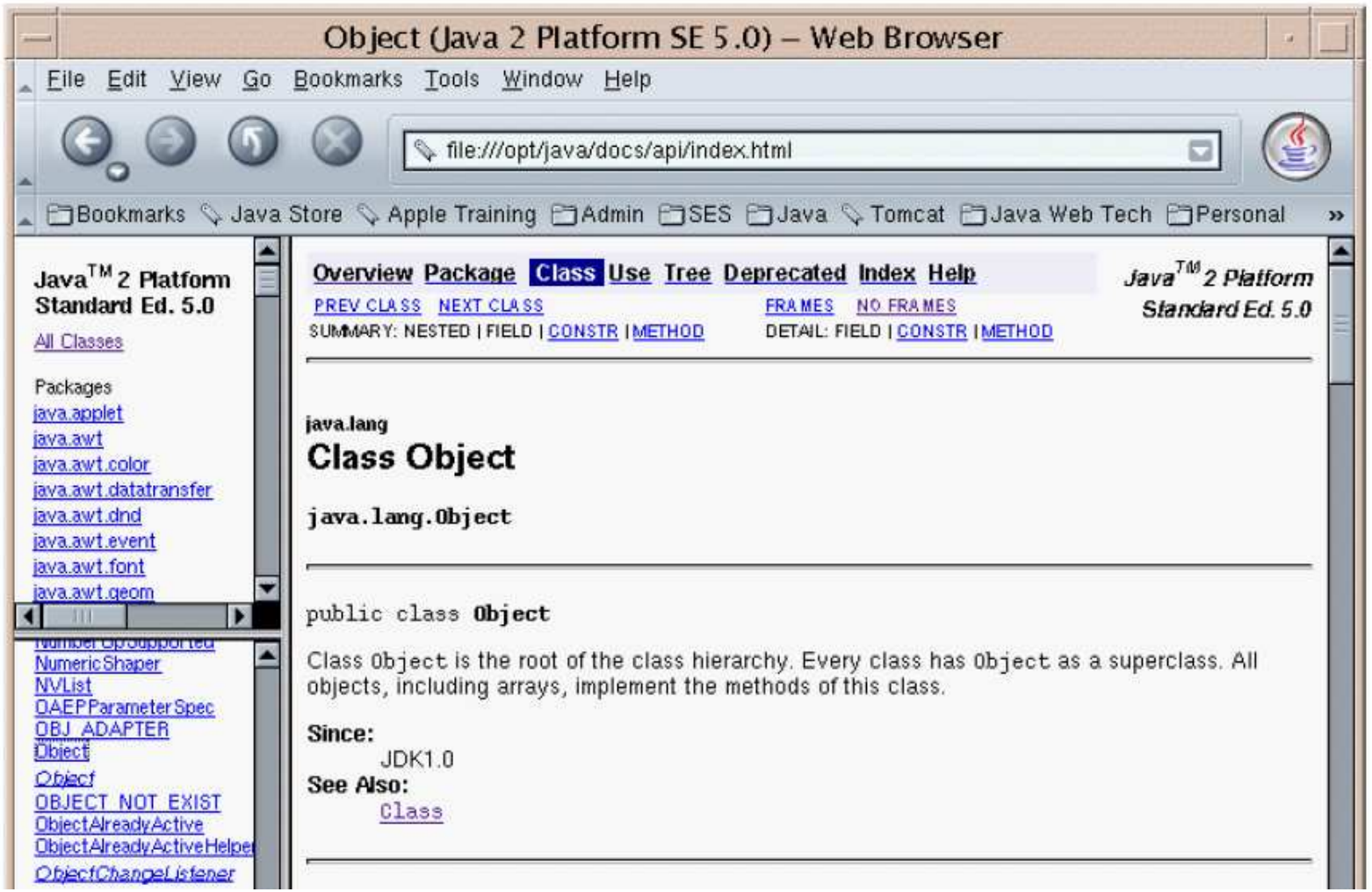
- Hilos
- Funciones de red



Comentarios

```
// comentario en una línea

/* comentario en una
 * o varias líneas
 */

/** el comentario de documentación
 *  también puede abarcar más de una línea
 */
```

Comentarios JavaDoc



Tipos primitivos

- Lógicos: boolean
- Textuales: char
- Enteros: byte, short, int y long
- Reales en coma flotante: double y float

```
boolean verdad = true;
```

```
'a'
'\t'
'\u????'
```

Tipo String

Los literales String se escriben entre comillas:

```
"El astuto zorro se lanza sobre el perro."
```

```
// declara e inicializa una variable char
char ch = 'A';

// declara dos variables char
char ch1,ch2;

// declara dos variables String y las inicializa
String greeting = "Buenos días \n";
String errorMessage = "No se ha encontrado el registro.";

// declara dos variables String
String str1,str2;
```

Enteros: byte, short, int y long

`2`	Éste es el formato decimal del entero 2.
`077`	El `0` inicial indica un valor octal.
`0xBAAC`	El prefijo `0x` indica que es un valor hexadecimal.

`2L`	La `L` indica que el valor decimal 2 se representa como un entero largo.
`077L`	El `0` inicial indica un valor octal.
`0xBAACL`	El prefijo `0x` indica que es un valor hexadecimal.

Longitud del entero	Nombre o tipo	Rango
8 bits	`byte`	De -2^7 a 2^7-1
16 bits	`short`	De -2^{15} a $2^{15}-1$
32 bits	`int`	De -2^{31} a $2^{31}-1$
64 bits	`long`	De -2^{63} a $2^{63}-1$

Reales en coma flotante: double y float

`3.14`	Valor en coma flotante sencillo (double) en notación estándar
`6.02E23`	Valor en coma flotante largo
`2.718F`	Valor corto de precisión simple (`float`)
`123.4E+306D`	Valor largo `double` con `D` redundante

Longitud	Nombre o tipo
32 bits	`float`
64 bits	`double`

- Los literales en coma flotante se consideran double a menos que se declaren expresamente como float
  - 2.718F
  - 2.718f

Declaraciones y asignaciones: ejemplo

```
1 public class Assign {
2   public static void main (String args []) {
3       // declara variables con tipos enteros
4       int x, y;
5       // declara y asigna un valor en coma flotante
6       float z = 3.414f;
7       // declara y asigna un valor de tipo double
8       double w = 3.1415;
9       // declara y asigna un valor boolean
10      boolean truth = true;
11      // declara una variable de tipo char
12      char c;
13      // declara una variable de tipo String
14      String str;
15      // declara y asigna un valor a una variable String
16      String str1 = "hasta la vista";
17      // asigna un valor a la variable char
18      c = 'A';
19      // asigna un valor a la variable String
20      str = "¡Hola!";
21      // asigna valores a las variables int
22      x = 6;
23      y = 1000;
24      }
25 }
```

Tipos referencia

```
1   public class MyDate {
2     private int day = 1;
3     private int month = 1;
4     private int year = 2000;
5     public MyDate(int day, int month, int year) { ... }
6     public String toString() { ... }
7   }
```

```
1    public class TestMyDate {
2      public static void main(String[] args) {
3        MyDate today = new MyDate(22, 7, 1964);
4      }
5    }
```

Referencia a un objeto MyDate

Construcción e inicialización

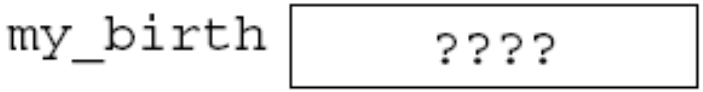
```
MyDate today = new MyDate(22, 7, 1964);
```

1. Se asigna e inicializa el espacio al nuevo objeto como 0 o vacío (null).
   - En programación Java, esta fase es indivisible para garantizar que no habrá objetos que contengan valores aleatorios.
2. Se realiza cualquier inicialización explícita.
3. Se ejecuta un *constructor*, un método especial.
   - Los argumentos de new se pasan al constructor (22, 7, 1964).
4. El valor resultante de new es una referencia al nuevo objeto en el espacio de memoria dinámica.
5. Esta referencia se guarda en la variable de referencia.

Construcción e inicialización

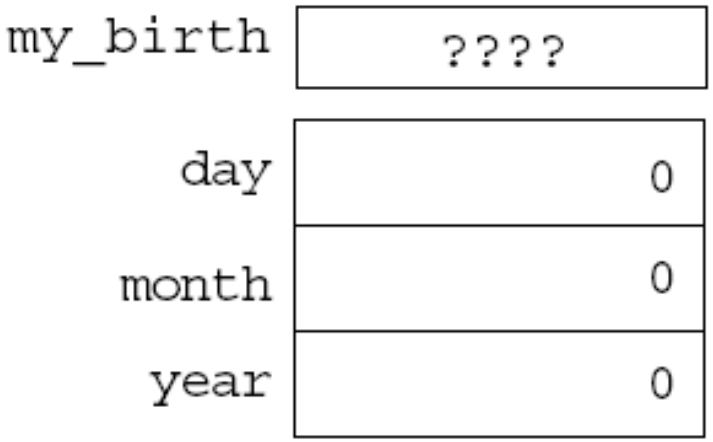
- 1º

```
MyDate my_birth = new MyDate(22, 7, 1964);
```



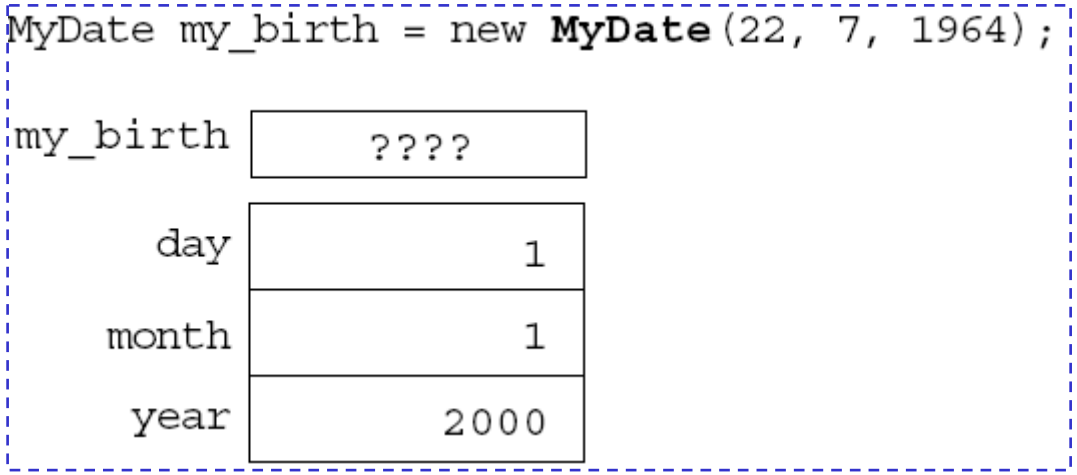
- 2º

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

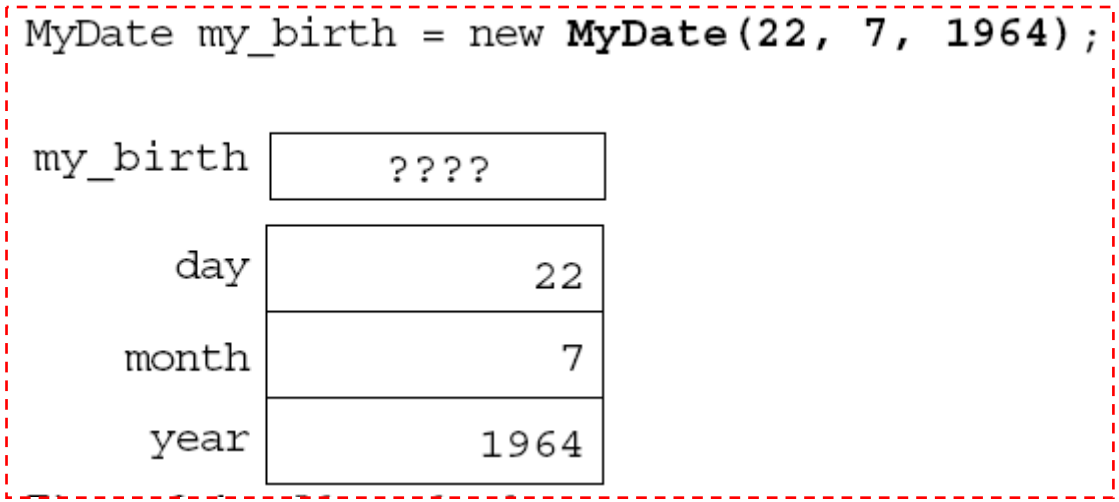


Construcción e inicialización

- 3º



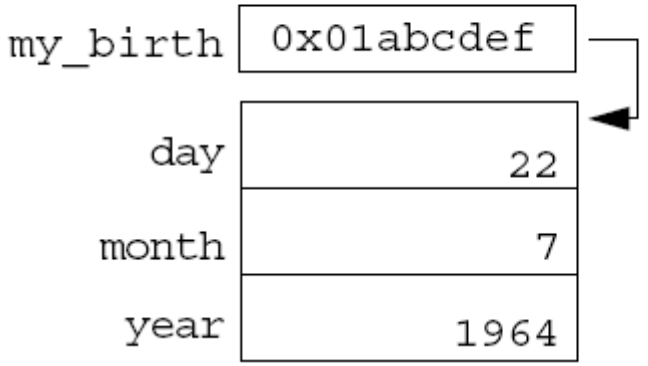
- 4º



Construcción e inicialización

- 5º

```
MyDate my_birth = new MyDate(22, 7, 1964);
```



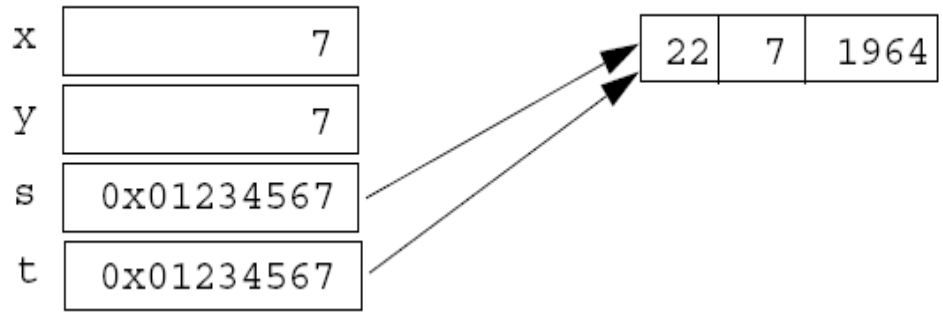
Uso de this

- Desambiguación
- Autoreferencia

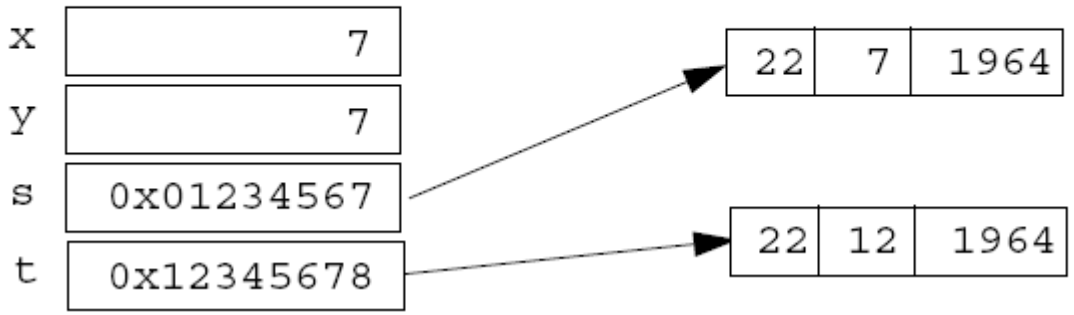
```
public class MyDate {
    private int day = 1;
    private int month = 1;
    private int year = 2000;

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public MyDate(MyDate date) {
        this.day = date.day;
        this.month = date.month;
        this.year = date.year;
    }

    public MyDate addDays(int moreDays) {
        MyDate newDate = new MyDate(this);
        newDate.day = newDate.day + moreDays;
        return newDate;
    }

    public String toString() {
        return "" + day + "/" + month + "/" + year;
    }
}
```

Asignación de referencias

```
int x = 7;
int y = x;
MyDate s = new MyDate(22, 7, 1964);
MyDate t = s;
```




```
t = new MyDate(22, 12, 1964); // reasigna la variable
```



Convenios de codificación en Java

- *Paquetes*: los nombres de paquetes se escriben en minúsculas.

```
package transporte.objetos
```

- *Clases*: sustantivos en mayúsculas y minúsculas; la primera letra de cada palabra en mayúscula

```
class LibroContabilidad
```

- *Interfaces*: mismas normas que las clases

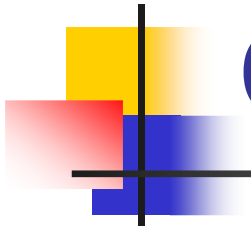
```
interface Contabilidad
```

- *Métodos*: verbos en mayúsculas y minúsculas, inicial en minúscula. Limitar el uso de signos de subrayado.

```
cuadrarLibro()
```

- *Variables*: en mayúsculas y minúsculas, con la inicial en minúscula. Limitar el uso de subrayado y evitar uso de dólar ($)

```
clienteActual
```



Convenios de codificación

- *Constantes*: en mayúsculas, separando palabras subrayado

```
TOTAL_NOMINA
CANTIDAD_MAXIMA
```

- *Estructuras de control*: if-else o for, escribirlas entre llaves ({ }), aunque sean sentencias sencillas.

```
if ( condición ) {
  sentencia1;
} else {
  sentencia2;
}
```

Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

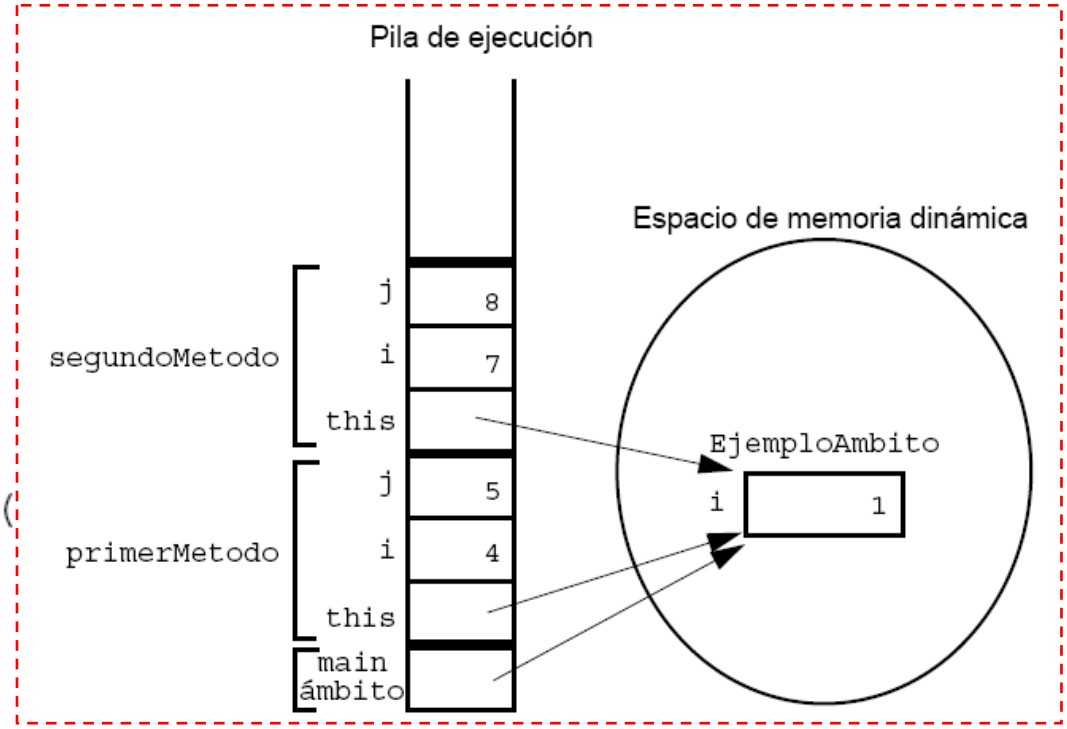
Desarrollo de interfaces gráficas de usuario

- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

Programación Java avanzada

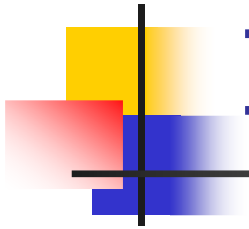
- Hilos
- Funciones de red

Ámbito de variables




```
1   public class EjemploAmbito
2     private int i=1;
3
4     public void primerMetodo(
5       int i=4, j=5;
6       this.i = i + j;
7       segundoMetodo(7);
8     }
9     public void segundoMetodo(int i) {
10      int j=8;
11      this.i = i + j;
12    }
13  }
```

```
1   public class PruebaAmbito {
2     public static void main(String[] args) {
3       EjemploAmbito ambito = new EjemploAmbito();
4       ambito.primerMetodo();
5     }
6   }
```



Inicialización de las variables

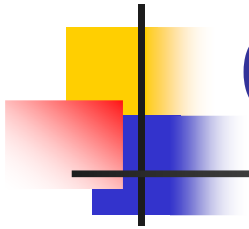
- Los atributos toman valores por defecto

Variable	Valor
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000'
boolean	false
Todos los tipos de referencia	null

- Las variables de métodos no
  - Hay que inicializarlas exprersamente

Operadores, precedencia

Precedencia	Operadores
D-I	++ -- + - ~ ! (<*tipo_dato*>)
I-D	* / %
I-D	+ -
I-D	<< >> >>>
I-D	< > <= >= instanceof
I-D	== !=
I-D	&
I-D	^
I-D	|
I-D	&&
I-D	||
D-I	<*expr_booleana*> ? <*expr1*> : <*expr2*>
D-I	= *= /= %= += -= <<= >>= >>>= &= ^= |=



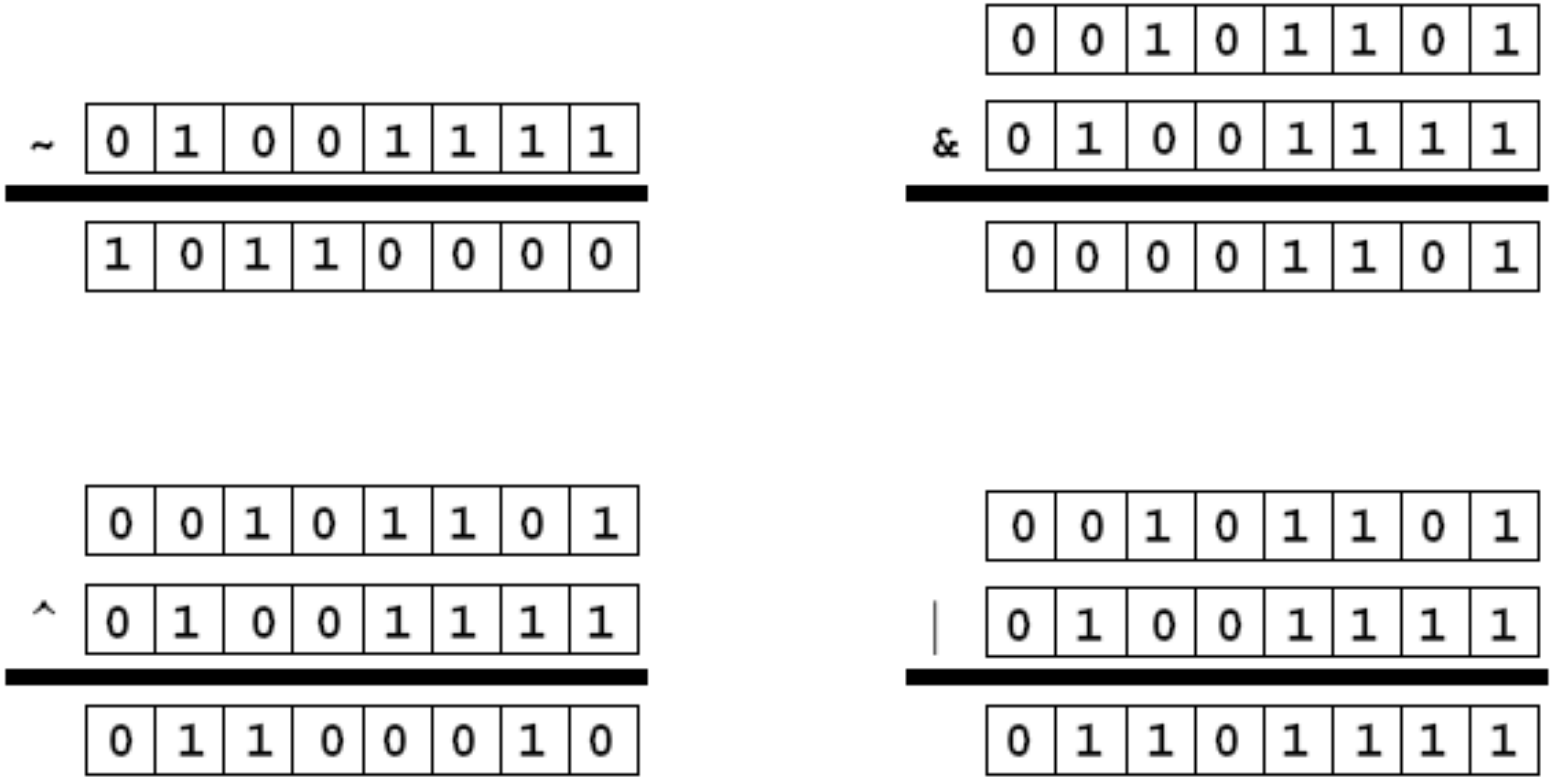
Operadores lógicos

```
int i = 1;
if (i)       // genera un error de compilación
if (i != 0) // Correcto
```

- Operan en cortocircuito

```
MiFecha d = reserva.getFechaSalida();
if ((d != null) && (d.day > 31)) {
  // hacer algo con d
}
```

Operadores lógicos de bits



Operadores de desplazamiento a la derecha >> y >>>

- >> aritmético o con signo
  - Copia el bit de signo al desplazar

```
128 >> 1 devuelve 128/2^1 = 64
256 >> 4 devuelve 256/2^4 = 16
-256 >> 4 devuelve -256/2^4 = -16
```

- >>> lógico o sin signo
  - Inserta 0 al desplazar

```
1010 ... >> 2 da como resultado 111010 ...
1010 ... >>> 2 da como resultado 001010 ...
```

- << a la izquierda

```
128 << 1 devuelve 128*2^1 = 256
16 << 2 devuelve 16*2^2 = 64
```

Solo se aplican a tipos enteros

>>> solo se aplica a int y long

Ejemplos de operadores de desplazamiento

Concatenación de cadenas: +

```
String saludo = "Dr. ";
String nombre = "Pedro" + " " + "Moreno";
String tratamiento = saludo + " " + nombre;
```



```
Dr. Pedro Moreno
```

Conversión de tipos

```
long valorGrande = 99L;
int squashed = valorGrande;          // Incorrecto, necesita
                                     //  conversión de tipos
int squashed = (int) valorGrande;    //  Correcto

int squashed = 99L;                  // Incorrecto, necesita
                                     //  conversión de tipos
int squashed = (int) 99L;            //  Correcto, pero...
int squashed = 99;                  //  literal entero
                                     //  predeterminado
```

Promoción y conversión de expresiones

```
long valgrande = 6;      // 6 es un tipo int, correcto
int valormenor = 99L;  // 99L es un tipo long, incorrecto

double z = 12.414F;  // 12.414F es un tipo float, correcto
float z1 = 12.414;   // 12.414 es double, incorrecto
```

- Todas las operaciones binarias dan como mínimo int, o el tipo más largo de los operandos

```
short a, b, c;
a = 1;
b = 2;
c = a + b;
```
<---- Error

```
c = (short)(a + b);
```
<---- Ok

Sentencias if, else

Correcto pero desaconsejado, no tiene llaves

```
if ( <expresión_booleana> )
  <sentencia_o_bloque>
```

```
if ( x < 10 )
  System.out.println("¿Ya has acabado?");
```

```
if ( x < 10 ) {
  System.out.println("¿Ya has acabado?");
}
```

```
if ( <expresión_booleana> )
  <sentencia_o_bloque>
else
  <sentencia_o_bloque>
```

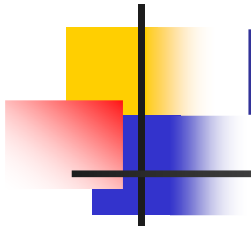
```
if ( x < 10 ) {
  System.out.println("¿Ya has acabado?");
} else {
  System.out.println("Sigo trabajando...");
}
```

Sentencia switch

```
switch ( <expresión> ) {
  case <constante1>:
    <sentencia_o_bloque>*
    [break;]
  case <constante2>:
    <sentencia_o_bloque>*
    [break;]
  default:
    <sentencia_o_bloque>*
    [break;]
}
```

```
switch ( modeloAutomovil) {
  case DELUXE:
    agregarAireAcondicionado();
    agregarRadio();
    agregarRuedas();
    agregarMotor();
    break;
  case ESTANDAR:
    agregarRadio();
    agregarRuedas();
    agregarMotor();
    break;
  default:
    agregarRuedas();
    agregarMotor();
}
```

```
switch ( modeloAutomovil) {
  case DELUXE:
    agregarAireAcondicionado();
  case ESTANDAR:
    agregarRadio();
  default:
    agregarRuedas();
    agregarMotor();
}
```

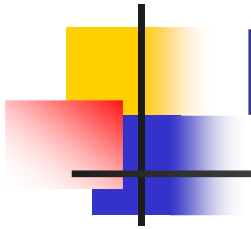


Bucles for

```
for ( <expr_inicial>; <expr_prueba>; <expr_alter> )
  <sentencia_o_bloque>
```

```
for ( int i = 0; i < 10; i++ ) {
  System.out.println(i + " al cuadrado es " + (i*i));
}
```

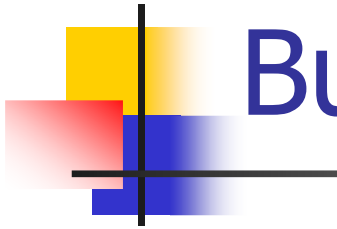
```
for (i = 0, j = 0; j < 10; i++, j++) { }
```



Bucle while

```
while ( <expr_prueba> )
   <sentencia_o_bloque>
```

```
int i = 0;
while ( i < 10 ) {
  System.out.println(i + " al cuadrado es " + (i*i));
  i++;
}
```



Bucle do/while

```
do
   <sentencia_o_bloque>
while ( <expr_prueba> );
```

```
int i = 0;
do {
   System.out.println(i + " al cuadrado es " + (i*i));
   i++;
} while ( i < 10 );
```

Construcciones especiales de control de bucles

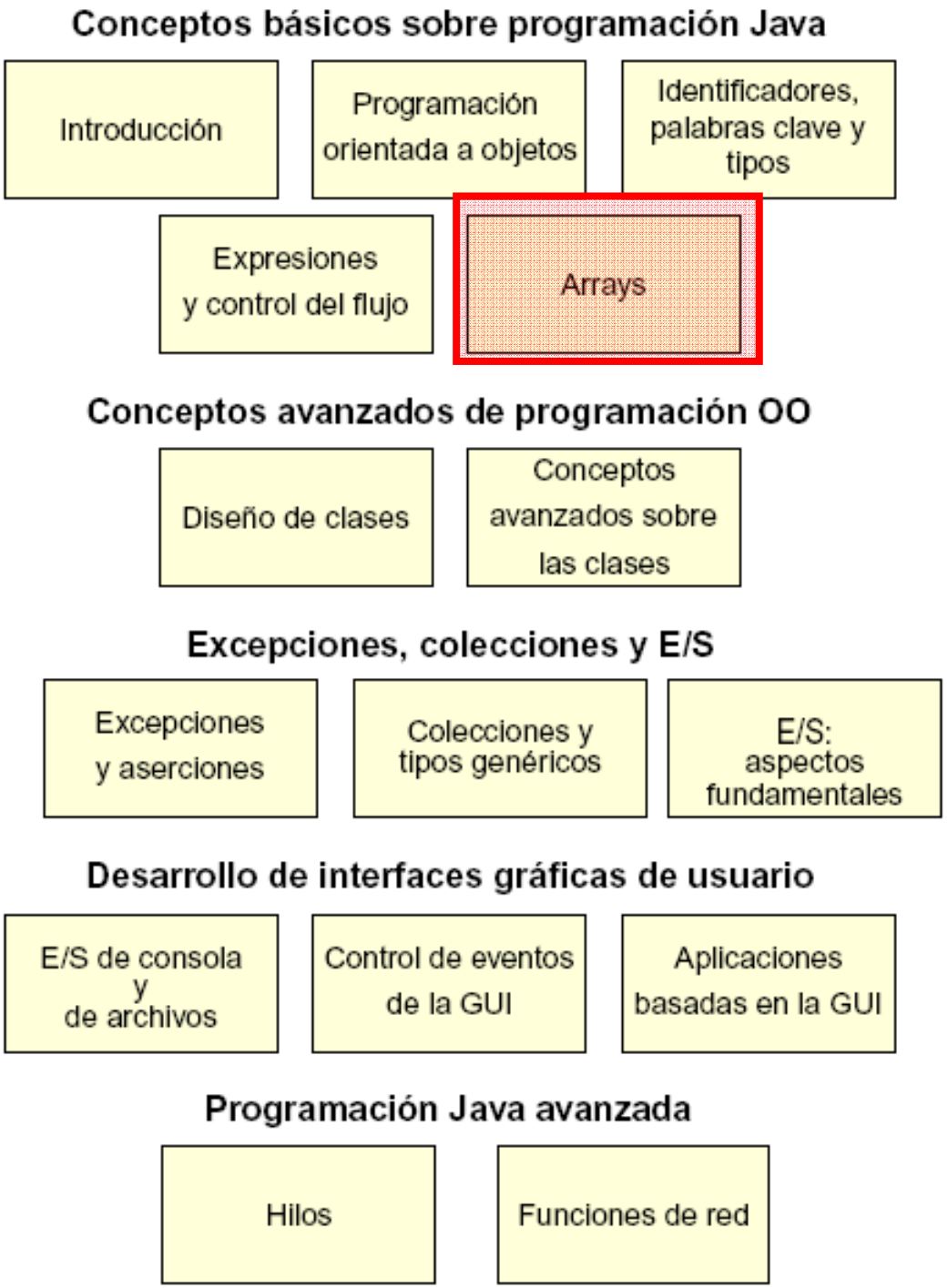
```
1    do {
2      sentencia;
3      if ( condición ) {
4        break;
5      }
6      sentencia;
7    } while ( expr_prueba );
```

```
1    do {
2      sentencia;
3      if ( condición ) {
4        continue;
5      }
6      sentencia;
7    } while ( expr_prueba );
```

```
1    outer:
2      do {
3        sentencia1;
4        do {
5          sentencia2;
6          if ( condición ) {
7            break outer;
8          }
9          sentencia3;
10       } while ( expr_prueba );
11       sentencia4;
12     } while ( expr_prueba );
```

```
1    test:
2      do {
3        sentencia1;
4        do {
5          sentencia2;
6          if ( condición ) {
7            continue test;
8          }
9          sentencia3;
10       } while ( expr_prueba );
11       sentencia4;
12     } while ( expr_prueba );
```

Contenidos


Conceptos básicos sobre programación Java
Introducción
Programación orientada a objetos
Identificadores, palabras clave y tipos
Expresiones y control del flujo
Arrays
Conceptos avanzados de programación OO
Diseño de clases
Conceptos avanzados sobre las clases
Excepciones, colecciones y E/S
Excepciones y aserciones
Colecciones y tipos genéricos
E/S: aspectos fundamentales
Desarrollo de interfaces gráficas de usuario
E/S de consola y de archivos
Control de eventos de la GUI
Aplicaciones basadas en la GUI
Programación Java avanzada
Hilos
Funciones de red

Declaración e inicialización de arrays

- De tipos primitivos o referencia
- Declaración

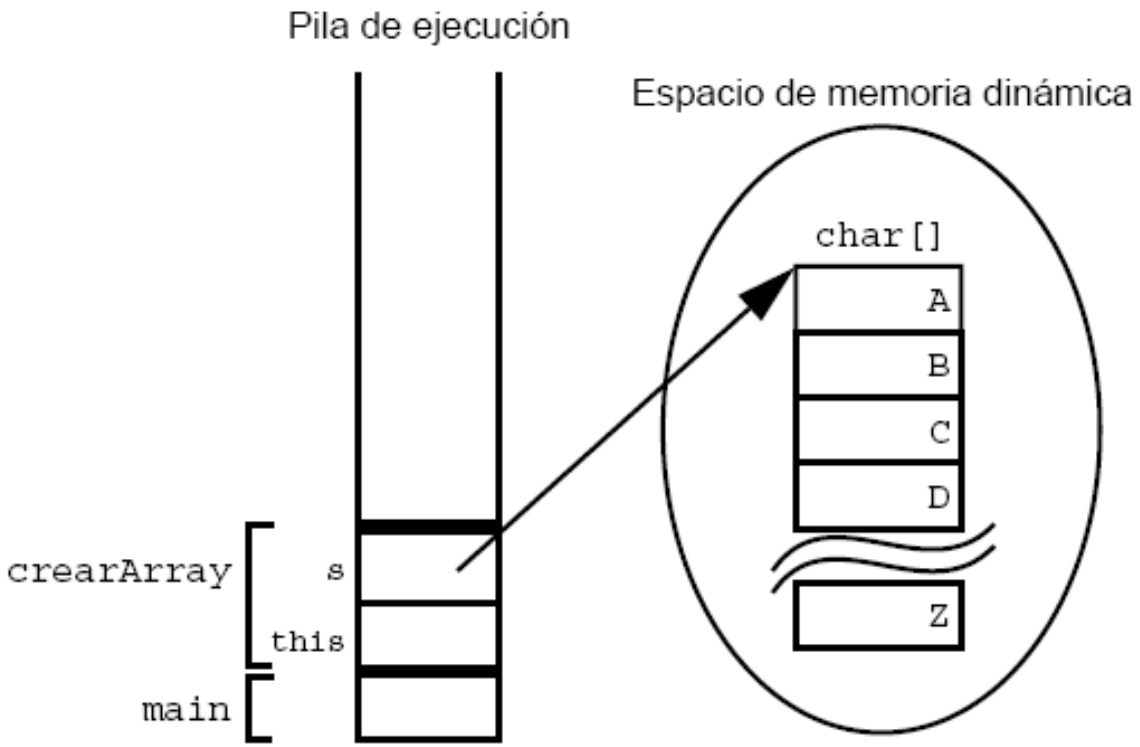
```
char[] s;
Point[] p; // donde Point es una clase
```

- Creación e Inicialización

```
s = new char[26];
```

```
1   public char[] crearArray() {
2     char[] s;
3
4     s = new char[26];
5     for ( int i=0; i<26; i++ ) {
6       s[i] = (char) ('A' + i);
7     }
8
9     return s;
10  }
```

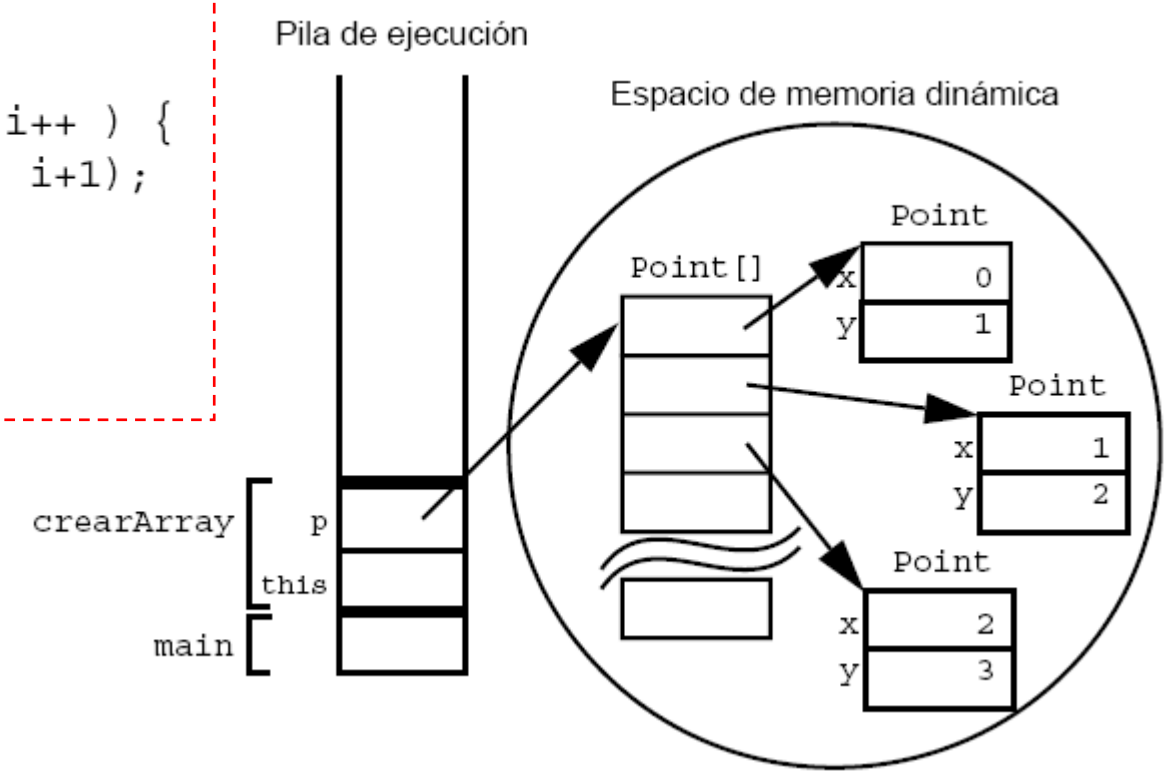
Arrays son objetos

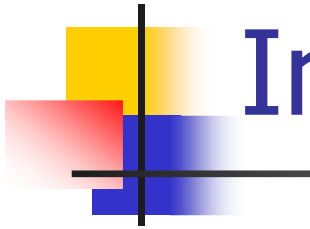


- Ubicados en espacio de memoria heap
- Alcanzados desde una variable referencia (s)

Arrays de referencias

```
public Point[] crearArray() {
  Point[] p;

  p = new Point[10];
  for ( int i=0; i<10; i++ ) {
    p[i] = new Point(i, i+1);
  }

  return p;
}
```





Inicialización

```
char[] s;
Point[] p; // donde Point es una clase
```

- Cada posición de s → '\u0000'
- Cada posición de Point → null

```
String[] names = {
  "Julia",
  "Juan",
  "Alfredo"
};
```

equivale a →

```
String[] names;
names = new String[3];
names[0] = "Julia";
names[1] = "Juan";
names[2] = "Alfredo";
```

Con tipos no básicos

```
MyDate[] dates = {
  new MyDate(22, 7, 1964),
  new MyDate(1, 1, 2000),
  new MyDate(22, 12, 1964)
};
```

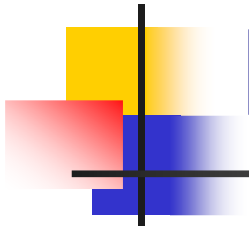
Arrays multidimensionales

```
int[][] dosDimen = new int [4][];
dosDimen[0] = new int[5];
dosDimen[1] = new int[5];
```

```
dosDimen[0] = new int[2];
dosDimen[1] = new int[4];
dosDimen[2] = new int[6];
dosDimen[3] = new int[8];
```

```
int[][] dosDimen = new int[4][5];
```

- Arrays de arrays
- Arrays dentados (o no rectangulares)
- Array rentangular



Límites de los arrays

- Rango desde 0..(tamaño – 1)
- Atributo length → tamaño del vector
- Si se sale de limites lanza OutOfBoundsException

```
public void printElements(int[] list) {
  for ( int i = 0; i < list.length; i++ ) {
    System.out.println(list[i]);
  }
}
```

Uso del bucle for mejorado

```
public void printElements(int[] list) {
  for ( int i = 0; i < list.length; i++ ) {
    System.out.println(list[i]);
  }
}
```

- Equivalentes

```
public void printElements(int[] list) {
  for ( int element : list ) {
    System.out.println(element);
  }
}
```

Copia de arrays

- Utilidad de copia en clase System

```
1    // array original
2    int[] miArray = { 1, 2, 3, 4, 5, 6 };
3
4    // nuevo array más largo
5    int[] copia = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
6
7    // copia todos los elementos de miArray en el array
8    // copia, empezando por el índice 0
9    System.arraycopy(miArray, 0, copia, 0,
10                        miArray.length);
```

Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

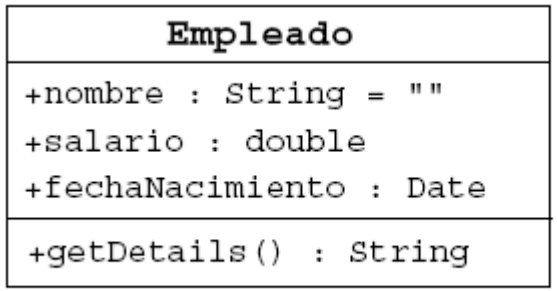
Desarrollo de interfaces gráficas de usuario

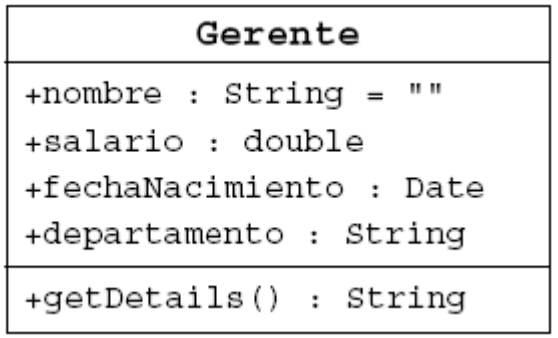
- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

Programación Java avanzada

- Hilos
- Funciones de red

Subclases

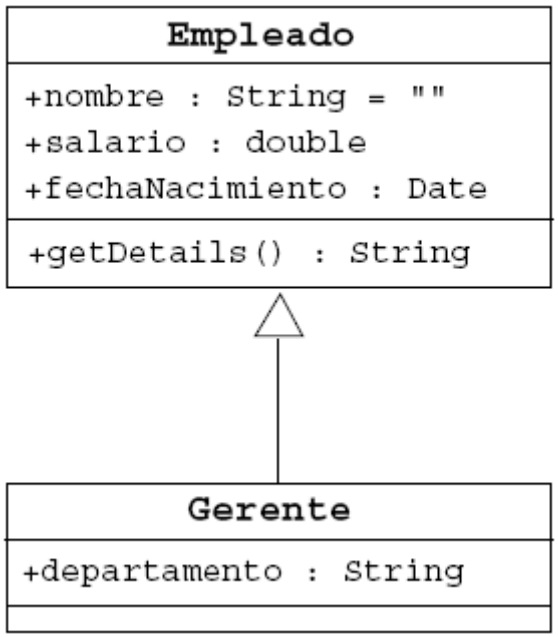





```
public class Empleado {
  public String nombre = "";
  public double salario;
  public Date fechaNacimiento;

  public String getDetails() {...}
}
```

```
public class Gerente {
  public String nombre = "";
  public double salario;
  public Date fechaNacimiento;
  public String departamento;

  public String getDetails() {...}
}
```

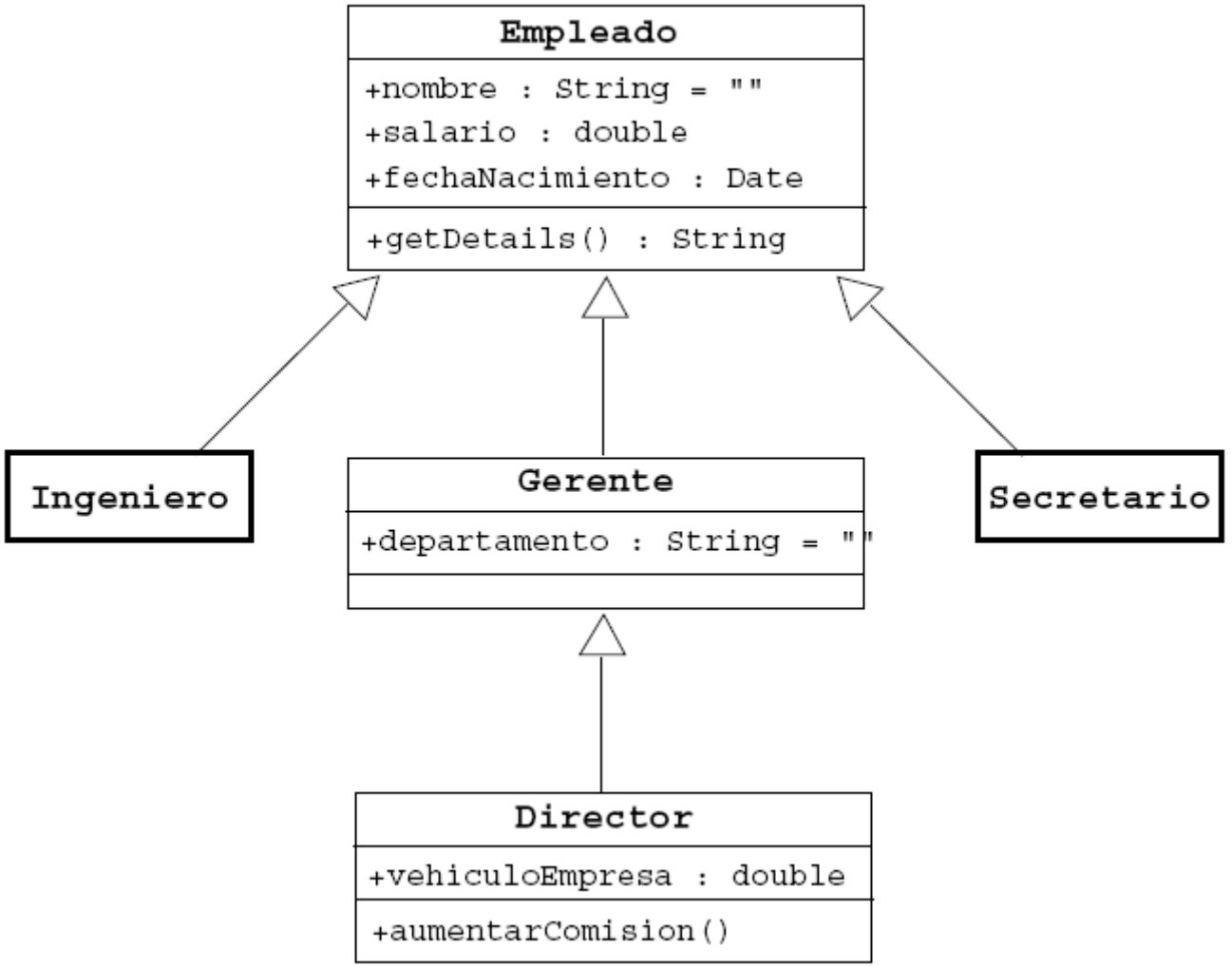
Subclases




```
public class Empleado {
  public String nombre = "";
  public double salario;
  public Date fechaNacimiento;

  public String getDetails() {...}
}
```

```
public class Gerente extends Empleado {
  public String departamento;
}
```

Herencia sencilla



Control de acceso

- Niveles de acceso

Modificador	La misma clase	El mismo paquete	Subclase	Universal
`private`	Sí			
predeterminado	Sí	Sí		
`protected`	Sí	Sí	Sí	
`public`	Sí	Sí	Sí	Sí

Sobrescritura de métodos (redefinición)

```
public class Empleado {
  protected String nombre;
  protected double salario;
  protected Date fechaNacimiento;

  public String getDetails() {
    return "Nombre: " + nombre + "\n"
         + "Salario: " + salario;
  }
}
```

- Modificación del comportamiento de la clase original

redefine

```
public class Gerente extends Empleado {
  protected String departamento;

  public String getDetails() {
    return "Nombre: " + nombre + "\n"
         + "Salario: " + salario + "\n"
         + "Gerente de: " + departamento;
  }
}
```

Redefinición

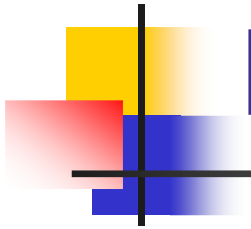
```
public class Padre {
  public void hacerAlgo() {}
}

public class Hijo extends Padre {
  private void hacerAlgo() {} // no válido
}
```

No se puede reducir la visibilidad

```
public class UsoAmbos {
  public void hacerOtraCosa() {
    Padre p1 = new Padre();
    Padre p2 = new Hijo();
    p1.hacerAlgo();
    p2.hacerAlgo();
  }
}
```

Llamada a métodos redefinidos

```
public class Empleado {
  private String nombre;
  private double salario;
  private Date fechaNacimiento;

  public String getDetails() {
    return "Nombre: " + nombre + "\nSalario: " + salario;
  }
}

public class Gerente extends Empleado {
  private String departamento;

  public String getDetails() {
    // llama al método de la superclase
    return super.getDetails()
           + "\nDepartamento: " + departamento;
  }
}
```



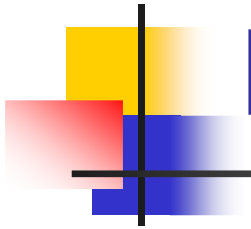
Polimorfismo

- Una variable referencia es polimórfica, puede "apuntar" a varios tipos de objetos

```
Empleado e = new Gerente(); //válido
```

- Sin embargo no es posible

```
// Intento no permitido
e.departamento = "Ventas";
```



Métodos virtuales

- ¿A qué método getDetails() se llama?
  - ¿Al de Gerente?
  - ¿Al de Empleado?

```
Empleado e = new Gerente();
e.getDetails();
```

- Se determina en runtime
  - Depende el objeto asociado y de la declaración del método
  - En java por defecto todos los método son virtuales
  - En este caso se ejecutaría el de Gerente

Argumentos polimórficos

```
public class ServicioImpuestos {
  public TipoImpositivo hallarTipoImpositivo(Empleado e) {
    // hace los cálculos y devuelve el tipo impositivo para e
  }
}
```

```
ServicioImpuestos svcImp = new ServicioImpuestos();
Gerente m = new Gerente();
TipoImpositivo t = svcImp.hallarTipoImpositivo(m);
```

Operador instanceof

Aceptable aunque redundante

```
public class Empleado extends Object
public class Gerente extends Empleado
public class Ingeniero extends Empleado
```

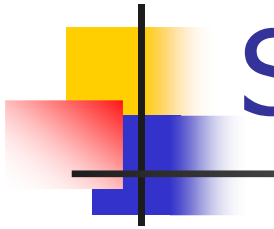
```
public void hacerAlgo(Empleado e) {
  if ( e instanceof Gerente ) {
    // Procesar un Gerente
  } else if ( e instanceof Ingeniero ) {
    // Procesar un Ingeniero
  } else {
    // Procesar cualquier otro tipo de Empleado
  }
}
```

Usando bien el polimorfismo es muy poco habitual hacer estos chequeos. A menudo indicación de un mal diseño

Conversión de objetos

```
public void hacerAlgo(Empleado e) {
  if ( e instanceof Gerente ) {
    Gerente m = (Gerente) e;
    System.out.println("Éste es el gerente de "
                       + m.getDepartamento());
  }
  // resto de la operación
}
```

- Upcast siempre permitidos
- Downcast se fuerza con ()
  - El compilador chequea si es posible
  - En runtime se verifica realmente, si no casa salta ClassCastException



Sobrecarga de métodos

```
public void println(int i)
public void println(float f)
public void println(String s)
```

- Reglas:
  - Las listas de argumentos *deben* ser diferentes
  - Los tipos de retorno *pueden* ser diferentes
    - Pero no es suficiente si es la única diferencia

Métodos con argumentos variables

```
public class Estadistica {
  public float average(int x1, int x2) {}
  public float average(int x1, int x2, int x3) {}
  public float average(int x1, int x2, int x3, int x4) {}
}

float gradePointAverage = stats.average(4, 3, 4);
float averageAge = stats.average(24, 32, 27, 18);
```

```
public class Estadistica {
  public float average(int... nums) {
    int sum = 0;
    for ( int x : nums ) {
      sum += x;
    }
    return ((float) sum) / nums.length;
  }
}
```

array de tipo int[]

Sobrecarga de constructores

```
1   public class Empleado {
2     private static final double SALARIO_BASE = 15000.00;
3     private String nombre;
4     private double salario;
5     private Date fechaNacimiento;
6
7     public Empleado(String nombre, double salario, Date FdeNac) {
8       this.nombre = nombre;
9       this.salario = salario;
10      this.fechaNacimiento = FdeNac;
11    }
12    public Empleado(String nombre, double salario) {
13      this(nombre, salario, null);
14    }
15    public Empleado(String nombre, Date FdeNac) {
16      this(nombre, SALARIO_BASE, FdeNac);
17    }
18    public Empleado(String nombre) {
19      this(nombre, SALARIO_BASE);
20    }
21    // más código de Empleado...
22  }
```

this en un constructor debe ser la primera línea

Más sobre constructores

- Los constructores no se heredan
- Se puede llamar a los constructores de las clases padre

```
1   public class Gerente extends Empleado {
2     private String departamento;
3
4     public Gerente(String nombre, double salario, String depto) {
5       super(nombre, salario);
6       departamento = depto;
7     }
8     public Gerente(String nombre, String depto) {
9       super(nombre);
10      departamento = depto;
11    }
12    public Gerente(String depto) { // Este código da error
13      departamento = depto;
14    }
15  }
```

No hay Constructor sin parámetros en la clase padre

Clase Object

```
public class Empleado {
  // más código aquí
}
```

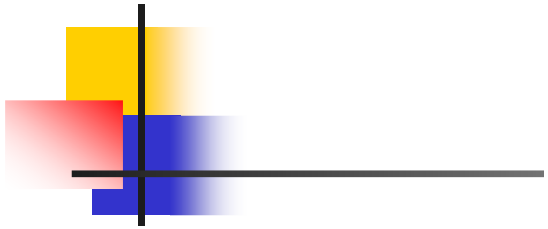
Equivale a:

```
public class Empleado extends Object {
  // más código aquí
}
```

- Equals
  - Distinto de ==
    - Equals equivalencia
    - == misma referencia
  - Se debe sobreescribir hashCode() a la vez
- toString
  - Devuelve representación en String de los valores del objeto

```
Date now = new Date();
System.out.println(now);
```

```
System.out.println(now.toString());
```



Equals y hashCode

```
1   public class MyDate {
2     private int day;
3     private int month;
4     private int year;
5
6     public MyDate(int day, int month, int year) {
7       this.day   = day;
8       this.month = month;
9       this.year  = year;
10    }
11
12    public boolean equals(Object o) {
13      boolean result = false;
14      if ( (o != null) && (o instanceof MyDate) ) {
15        MyDate d = (MyDate) o;
16        if ( (day == d.day) && (month == d.month)
17             && (year == d.year) ) {
18          result = true;
19        }
20      }
21      return result;
22    }
23
24    public int hashCode() {
25      return (day ^ month ^ year);
26    }
27  }
```

```
int pInt = 420;
Integer wInt = new Integer(pInt); // esto se denomina boxing
int p2 = wInt.intValue(); // esto se denomina unboxing
```

Clases envoltorio

- Java no considera los tipos de datos primitivos como objetos

Tipo de datos primitivos	Clase envoltorio
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

De String a int

```
int x = Integer.parseInt(str);
```

Conversión automática: autoboxing

- Boxing, unboxing

```
int pInt = 420;
Integer wInt = new Integer(pInt); // esto se denomina boxing
int p2 = wInt.intValue(); // esto se denomina unboxing
```

- Autoboxing, autounboxing

```
int pInt = 420;
Integer wInt = pInt; // esto se denomina autoboxing
int p2 = wInt; // esto se denomina autounboxing
```

Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

Desarrollo de interfaces gráficas de usuario

- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

Programación Java avanzada

- Hilos
- Funciones de red

Palabra clave static

- *static* declara miembros (atributos, métodos y clases anidadas) que están asociados a una clase en vez de a una instancia de la clase.

```
1   public class Counter {
2     private int serialNumber;
3     public static int counter = 0;
4
5     public Counter() {
6       counter++;
7       serialNumber = counter;
8     }
9   }
```

```
1   public class OtraClase {
2     public void incrementNumber() {
3       Count1.counter++;
4     }
5   }
```

Si no es private

Métodos de clase static

```
1   public class Count2 {
2     private int serialNumber;
3     private static int counter = 0;
4
5     public static int getTotalCount() {
6       return counter;
7     }
8
9     public Count2() {
10      counter++;
11      serialNumber = counter;
12    }
```

```
1   public class Count3 {
2     private int serialNumber;
3     private static int counter = 0;
4
5     public static int getSerialNumber() {
6       return serialNumber;  // ERROR DE
7     }
8   }
```

Error de compilación, atributo no estático

```
1   public class TestCounter {
2     public static void main(String[] args) {
3       System.out.println("El número del contador es "
4                          + Count2.getTotalCount());
5       Count2 count1 = new Count2();
6       System.out.println("El número del contador es "
7                          + Count2.getTotalCount());
8     }
9   }
```

Inicializadores estáticos

```
1   public class Count4 {
2     public static int counter;
3     static {
4       counter = Integer.getInteger("myApp.Count4.counter").intValue();
5     }
6   }
```

Se ejecutan al cargar la clase

Propiedad del sistema

```
1   public class TestStaticInit {
2     public static void main(String[] args) {
3       System.out.println("counter = "+ Count4.counter);
4     }
5   }
```

Ejecución:

```
java -DmyApp.Count4.counter=47 TestStaticInit
counter = 47
```

Palabra clave final

- Aplica a clases, métodos y variables
  - Clases: No es posible crear subclases
  - Métodos: No se puede redefinir
    - No son métodos virtuales, permite optimización en tiempo de ejecución
  - Variable: Se comporta como una constante

```
public class Banco {
  private static final
double  TASA_INTERES_PREDETERMINADA=3.2;
  // más declaraciones
}
```

Variables finales vacías

```
public class Customer {
  private final long customerID;

  public Customer() {
    customerID = createID();
  }
  public long getID() {
    return customerID;
  }
  private long createID() {
    return ... // genera nuevo ID
  }
  ... // más declaraciones
}
```

- Solo pueden asignarse una vez y ...
  - Si es atributo, en el constructor de la clase
  - Si es variable de método, en cualquier parte del método pero solo una vez

Tipos enumerados

```
1   package naipes.domain;
2
3   public enum Palo {
4      PICAS,
5      CORAZONES,
6      TREBOLES,
7      DIAMANTES
8   }
```

```
public class NaipeBaraja {
    private Palo palo;
    private int rango;

    public NaipeBaraja(Palo palo, int rango) {
        this.palo = palo;
        this.rango = rango;
    }

    public Palo getPalo() {return palo;}

    public String getNamePalo() {
        String name = "";
        switch ( palo ) {
            case PICAS:
                name = "Picas";
                break;
            case CORAZONES:
                name = "Corazones";
                break;
            case TREBOLES:
                name = "Treboles";
                break;
            case DIAMANTES:
                name = "Diamantes";
                break;
        }
        return name;
    }
}
```

Tipos enumerados avanzados

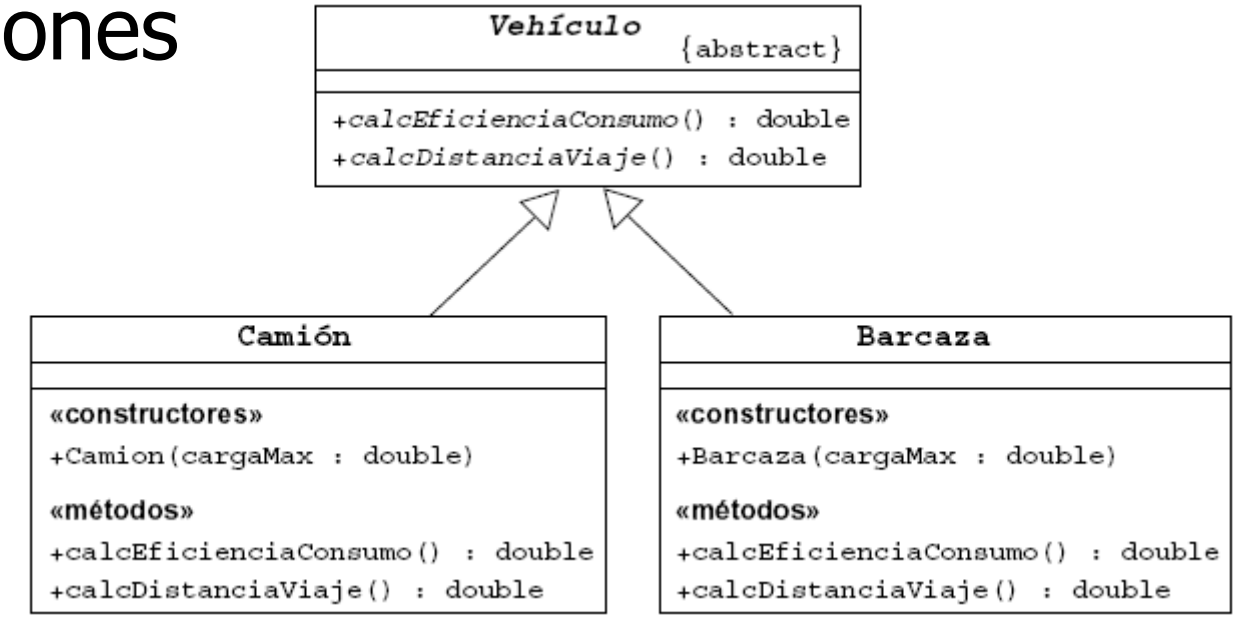
```
1   package naipes.tests;
2
3   import naipes.domain.NaipeBaraja;
4   import naipes.domain.Palo;
5
6   public class TestNaipeBaraja {
7     public static void main(String[] args) {
8
9       NaipeBaraja naipe1
10        = new NaipeBaraja(Palo.PICAS, 2);
11      System.out.println("el naipe1 es: " + naipe1.getRango()
12                         + " de " + naipe1.getPalo().getName());
13
14      // NuevoNaipeBaraja naipe2 = new NuevoNaipeBaraja(47, 2);
15      // Esto no se compilará.
16    }
17  }
```

```
1   package naipes.domain;
2
3   public enum Palo {
4     PICAS    ("Picas"),
5     CORAZONES   ("Corazones"),
6     TREBOLES    ("Treboles"),
7     DIAMANTES ("Diamantes");
8
9     private final String name;
10
11    private Palo(String name) {
12      this.name = name;
13    }
14
15    public String getName() {
16      return name;
17    }
18  }
```

- Se permite añadir métodos
- El constructor es privado

Clases abstractas

- La clase base indica que operaciones deben soportar las subclases pero no tiene conocimiento para resolver las operaciones



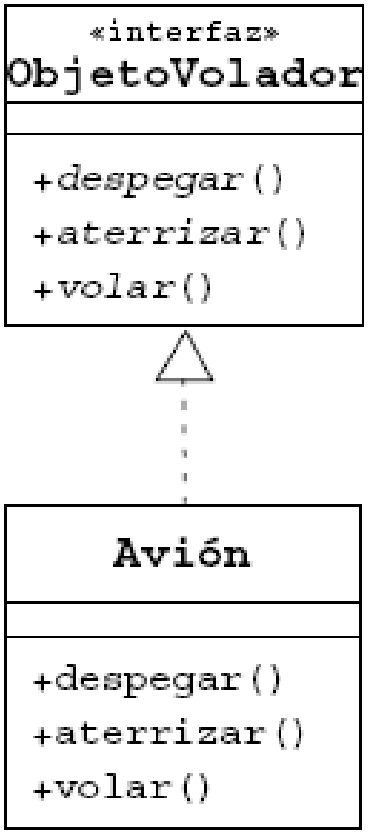
Clases abstractas

```
1   public abstract class Vehiculo {
2     public abstract double calcEficienciaConsumo();
3     public abstract double calcDistanciaViaje();
4   }
```

```
1   public class Camion extends Vehiculo {
2     public Camion(double cargaMax) {...}
3     public double calcEficienciaConsumo() {
4       // calcula el consumo de combustible de un camión
5     }
6     public double calcDistanciaViaje() {
7       // calcula la distancia de este viaje por autopista
8     }
9   }
```

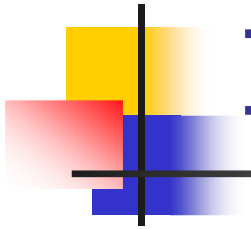
```
1   public class Barcaza extends Vehiculo {
2     public Barcaza(double cargaMax) {...}
3     public double calcEficienciaConsumo() {
4       // calcula el consumo de combustible de una barcaza
5     }
6     public double calcDistanciaViaje() {
7       // calcula la distancia de este viaje por vías fluviales
8     }
9   }
```

Interfaces



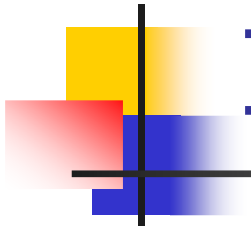

```
1   public interface ObjetoVolador {
2     public void despegar();
3     public void aterrizar();
4     public void volar();
5   }
```

```
1   public class Avion implements ObjetoVolador {
2     public void despegar() {
3       // acelerar hasta despegar
4       // subir el tren de aterrizaje
5     }
6     public void aterrizar() {
7       // bajar el tren de aterrizaje
8       // decelerar y desplegar flaps hasta tocar tierra
9       // frenar
10    }
11    public void volar() {
12      // mantener los motores en marcha
13    }
14  }
```



Interfaces

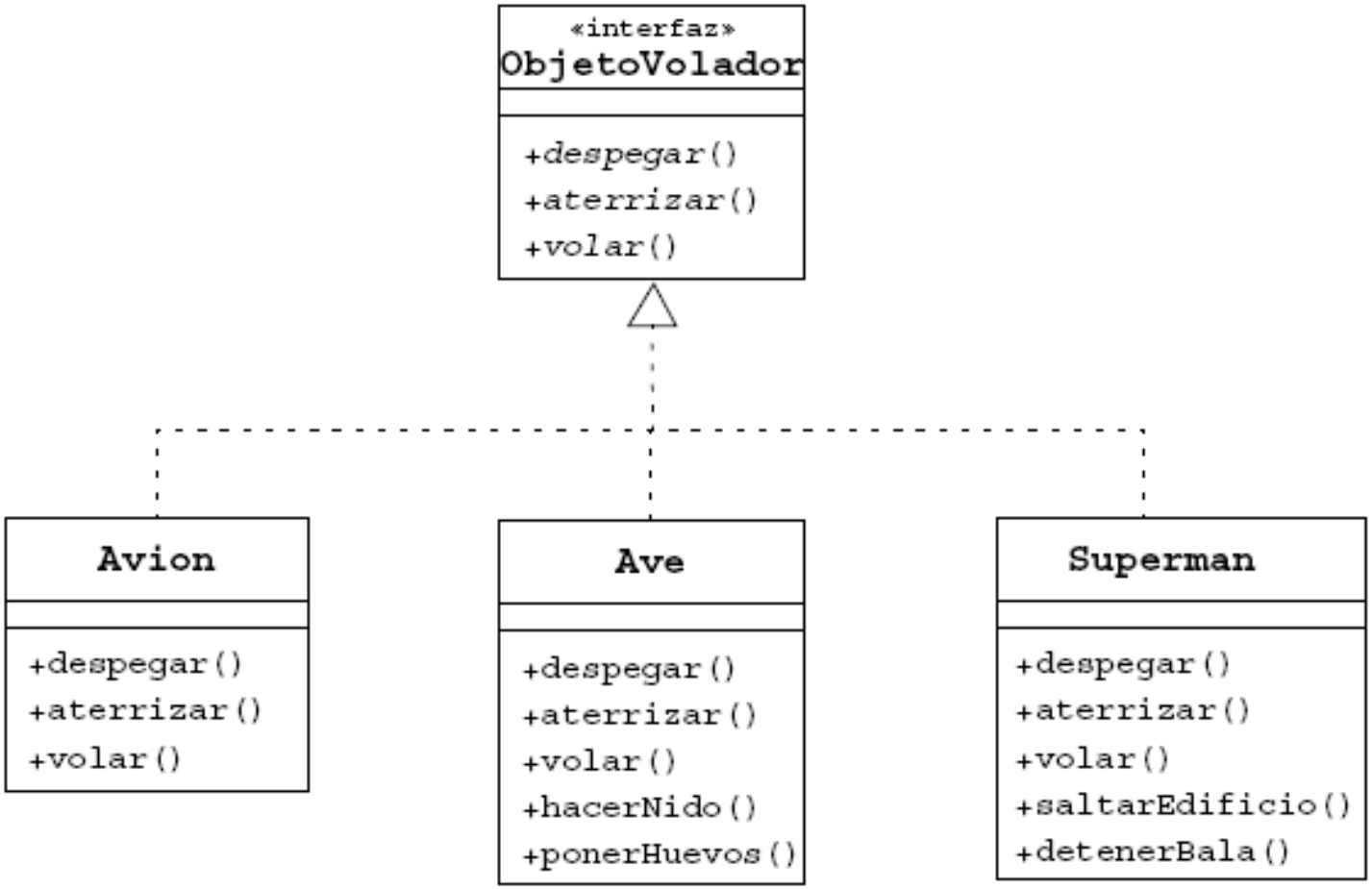
- Definen un contrato entre código cliente y la clase que proporciona la implementación
  - Una clase implementa un interfaz definiendo todos sus métodos
  - Muchas clases pueden implementar un interfaz sin que haya jerarquía entre ellas
  - Una clase puede implementar varios interfaces → herencia múltiple de interfaces
  - Los nombres de interfaz son tipos de variables referencia
  - Conversiones de tipos e `instanceOf`



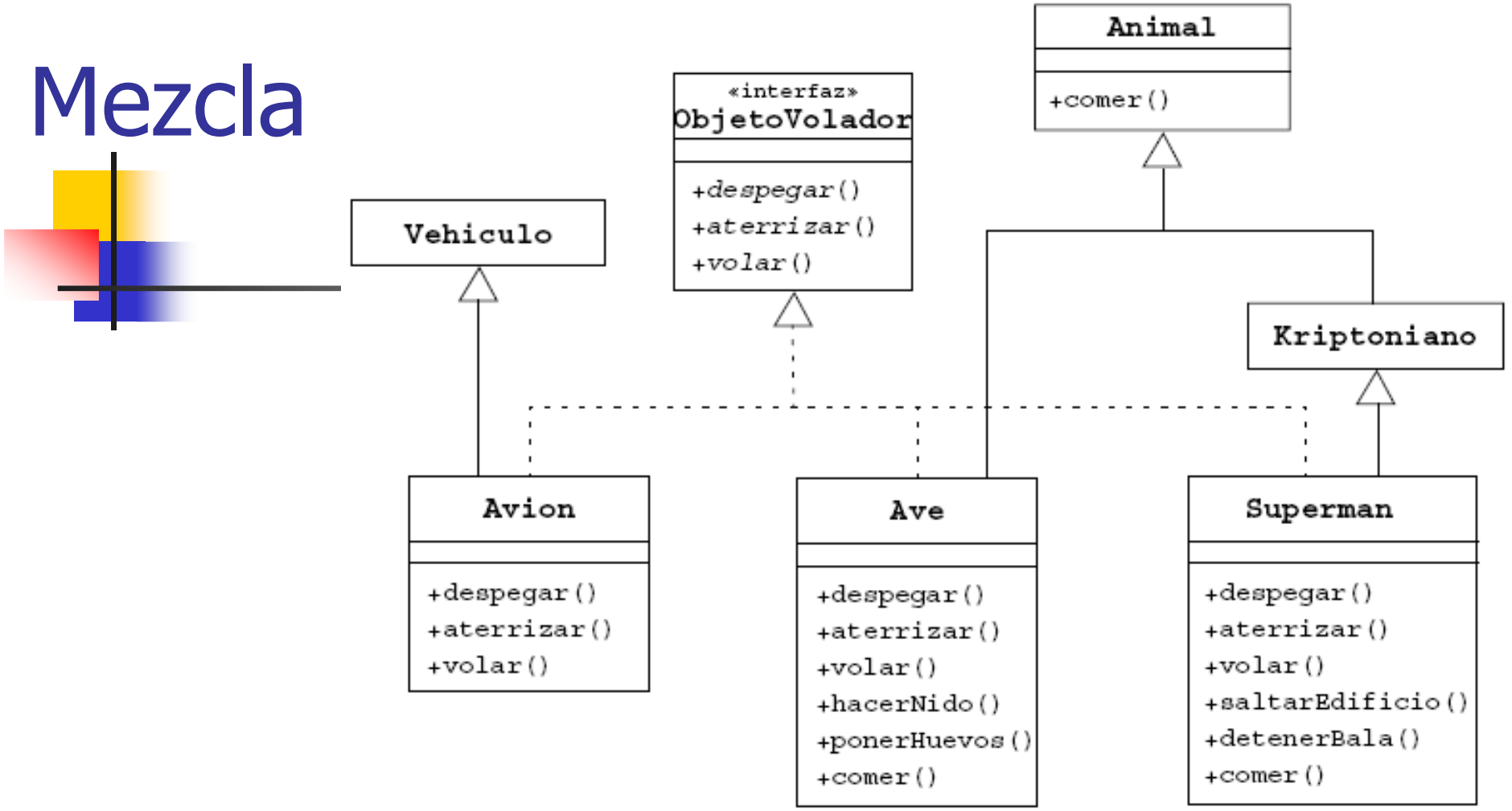
Interfaces

- Sin necesidad de indicarlo:
  - Todos los métodos declarados en una interfaz son públicos y abstract
  - Todos los atributos son public, static y final → constantes

Múltiples implementaciones

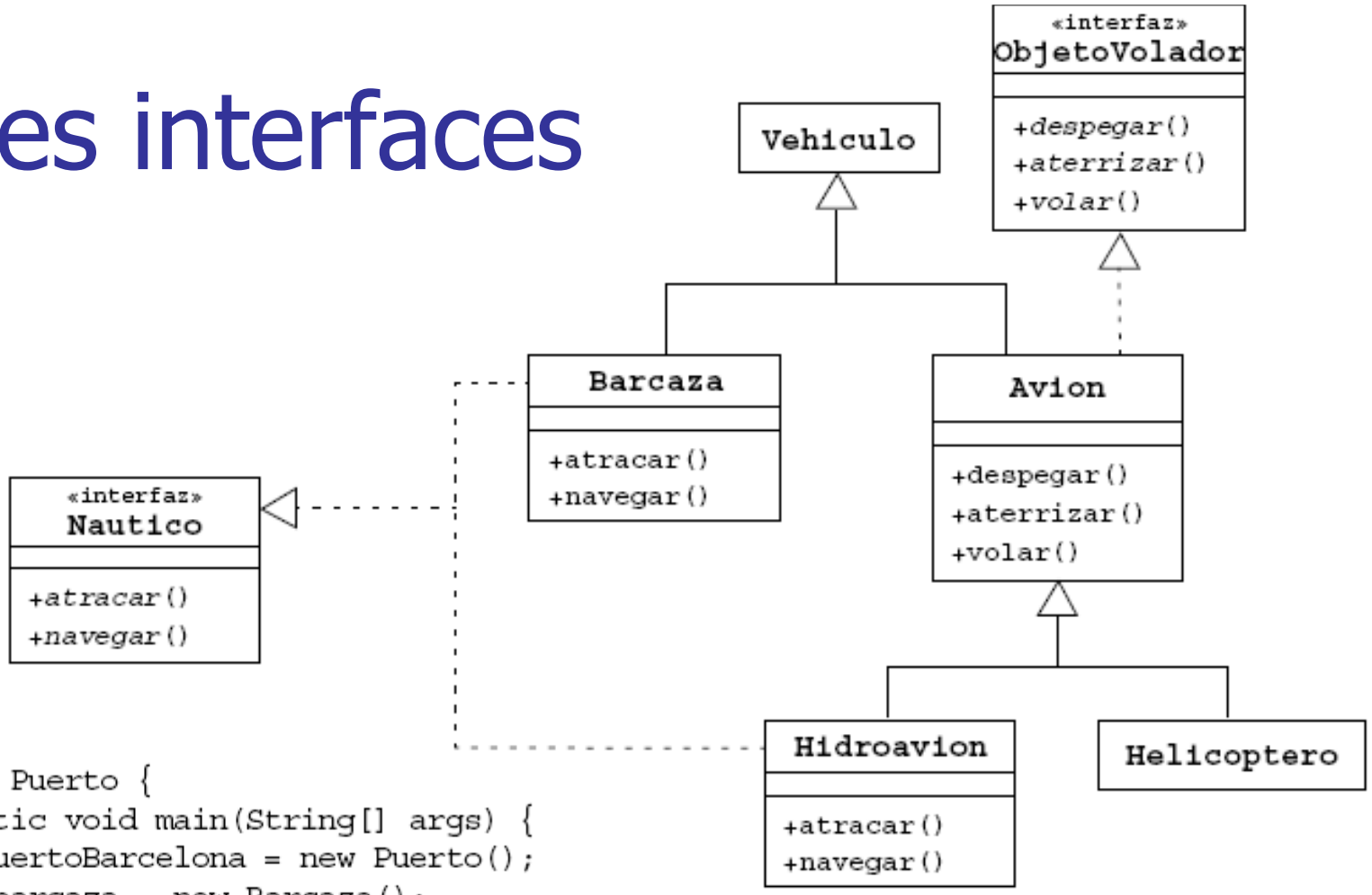


Mezcla

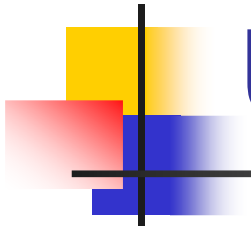



```
public class Ave extends Animal implements ObjetoVolador {
  public void despegar()  { /* implementación de despegar   */ }
  public void aterrizar()     { /* implementación de aterrizar   */ }
  public void volar()       { /* implementación de volar  */ }
  public void hacerNido(){ /* comportamiento de nidificación   */ }
  public void ponerHuevos()  { /* implementación de puesta de huevos  */ }
  public void comer()  { /* sobrescritura de la acción de comer  */ }
}
```

Múltiples interfaces




```
1   public class Puerto {
2     public static void main(String[] args) {
3       Puerto puertoBarcelona = new Puerto();
4       Barcaza barcaza = new Barcaza();
5       Hidroavion hAvion = new Hidroavion();
6
7       puertoBarcelona.darPermisoAtracar(barcaza);
8       puertoBarcelona.darPermisoAtracar(hAvion);
9     }
10    private void darPermisoAtracar(Nautico n) {
11      n.atracar();
12    }
13  }
```



Usos de las interfaces

- Declarar métodos que serán implementados por una o varias clases
- Dar a conocer la interfaz de programación de un objeto sin revelar el verdadero cuerpo de la clase (esto puede ser útil al distribuir un paquete de clases a otros desarrolladores)
- Identificar las similitudes entre clases no relacionadas sin tener que establecer ninguna relación entre ellas
- Simular la herencia múltiple declarando una clase que implemente varias interfaces

Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

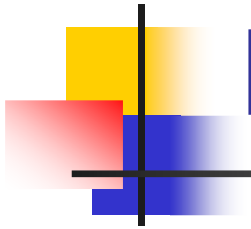
- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

Desarrollo de interfaces gráficas de usuario

- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

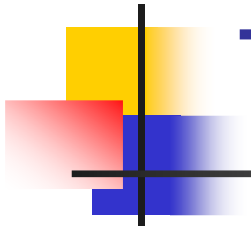
Programación Java avanzada

- Hilos
- Funciones de red



Excepciones y aserciones

- Excepciones: mecanismo de gestión de errores
  - Muy común en los lenguajes OO
- Aserciones: forma de verificar ciertos supuestos sobre la lógica de un programa
  - Pueden suprimirse al ejecutar el código (las excepciones no)



Tipos de Excepciones

- Chequeadas
  - Indican situaciones de error que se espera que el programador gestione (escriba código para ello)
- No chequeadas
  - Indican errores en el código o el entorno de imposible o difícil recuperación → no se obliga al programador a escribir código para gestionarlas
  - Errores: subtipo de no chequeadas, indican problemas en la JVM (OutOfMemoryError, etc)

Excepciones, ejemplo

```
1   public class AddArguments {
2     public static void main(String args[]) {
3       int sum = 0;
4       for ( int i = 0; i < args.length; i++ ) {
5         sum += Integer.parseInt(args[i]);
6       }
7       System.out.println("Sum = " + sum);
8     }
9   }
```

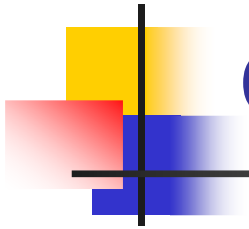
```
java AddArguments 1 2 3 4
Sum = 10

java AddArguments 1 dos 3.0 4
Exception in thread "main" java.lang.NumberFormatException:
For input string: "dos"at
java.lang.NumberFormatException.forInputString(NumberFormatE
xception.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at AddArguments.main(AddArguments.java:5)
```

Sentencia try-catch

```
1   public class AddArguments2 {
2     public static void main(String args[]) {
3       try {
4         int sum = 0;
5         for ( int i = 0; i < args.length; i++ ) {
6           sum += Integer.parseInt(args[i]);
7         }
8         System.out.println("Sum = " + sum);
9       } catch (NumberFormatException nfe) {
10        System.err.println("Uno de los argumentos de la línea de "
11                           + "comandos no es un entero.");
12      }
13    }
14  }
```

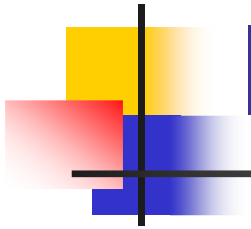
```
java AddArguments2 1 dos 3.0 4
Uno de los argumentos de la línea de comandos no es un entero.
```

Uso de múltiples cláusulas catch



```
try {
  // código que podría generar una o varias excepciones

} catch (MiException e1) {
  // código que debe ejecutarse si se envía MiException

} catch (MiOtraException e2) {
  // código que debe ejecutarse si se envía MiOtraException

} catch (Exception e3) {
  // código que debe ejecutarse si se envía cualquier otra
  // excepción
}
```

Mecanismo de apilamiento de llamadas



- Si una sentencia envía una excepción que no se gestiona en el método que la rodea, la excepción se envía al método que ha efectuado la llamada
- Si tampoco la gestiona el método de llamada, se vuelve a enviar al método inmediatamente anterior, y así sucesivamente.
- Si, cuando la excepción llega al método main(), éste tampoco la gestiona, la excepción interrumpe el programa de forma irregular.

Propagación de la excepción: ejemplo

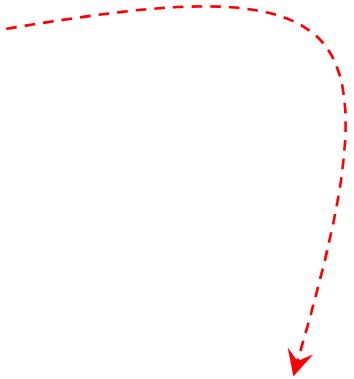
```
public class LanzaExcepcion {

    public static void main(String[] args) throws Exception {
        primero();
    }

    private static void primero() throws Exception {
        segundo();
    }

    private static void segundo() throws Exception {
        tercero();
    }

    private static void tercero() throws Exception {
        throw new Exception();
    }
}
```

Ejecución

```
Exception in thread "main" java.lang.Exception
        at LanzaExcepcion.tercero(LanzaExcepcion.java:17)
        at LanzaExcepcion.segundo(LanzaExcepcion.java:13)
        at LanzaExcepcion.primero(LanzaExcepcion.java:9)
        at LanzaExcepcion.main(LanzaExcepcion.java:5)
```

Propagación: otro ejemplo

```
public class LanzaExcepcion {

    public static void main(String[] args) {
        primero();
    }

    private static void primero() {
        try {
            segundo();
        } catch (Exception e) {
            System.out.println("Excepcion capturada");
        }
    }

    private static void segundo() throws Exception {
        tercero();
    }

    private static void tercero() throws Exception {
        throw new Exception();
    }
}
```



Ejecución

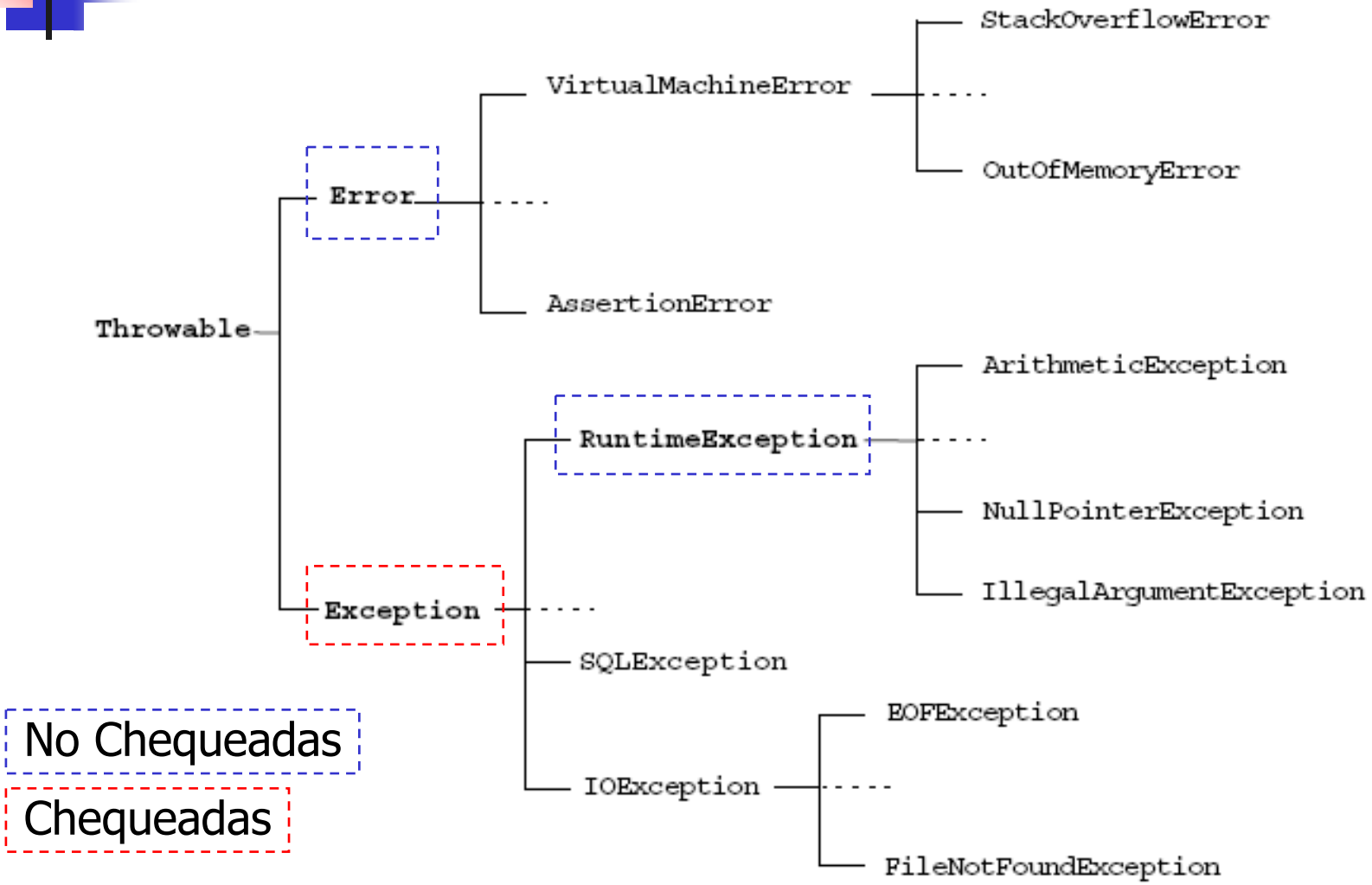
```
Excepcion capturada
```

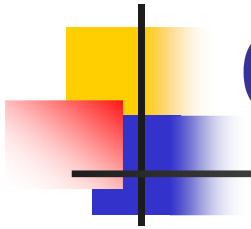
Cláusula finally

```
public static void main(String[] args) {
    try {
        abrirFichero();
        tratarFichero();
    }
    catch(ProcessingException pex){
        // manage exception...
    }
    finally{
        cerrarFichero();
    }
}

private static void tratarFichero() throws ProcessingException {
    // ...
    if (errorInProcess()){
        throw new ProcessingException("Error in ...");
    }
    // ...
}
```

- Garantiza que el código de su bloque siempre se ejecuta, incluso si se lanza excepción
- No detiene la propagación de la excepción

Categorías de excepciones





Categorías de excepciones

- Error:
  - Indica problemas en la JVM, no son errores recuperables. La JVM debe parar.
- RuntimeException:
  - Errores de consistencia del programa, generalmente por errores de programación. El programa no se suele poder recuperar.
- Exception:
  - Problemas en tiempo de ejecución causados por efectos del entorno y suelen poder gestionarse

La regla de la gestión o la declaración

- Aplica a excepciones chequeadas
- Si un bloque de código lanza una excepción:
  - O la gestiona

    try-catch-finally
  - O declara que la lanza

```
void trouble() throws IOException, OtraException {...}
```

Sobrescritura de métodos y excepciones

```
1   public class PruebaA {
2     public void metodoA() throws IOException {
3       // realiza algún cálculo
4     }
5   }
```

```
1   public class PruebaB1 extends PruebaA {
2     public void metodoA() throws EOFException {
3       // realiza algún cálculo
4     }
5   }
```

Es hija de IOException

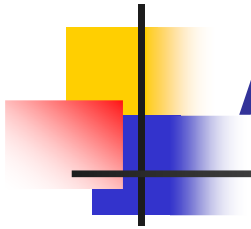
```
1   public class PruebaB2 extends PruebaA {
2     public void metodoA() throws Exception {
3       // realiza algún cálculo
4     }
5   }
```

No compila, Exception no es hija de IOException

Creación de excepciones personalizadas

```
1   public class ServerTimedOutException extends Exception {
2     private int port;
3
4     public ServerTimedOutException(String message, int port) {
5       super(message);
6       this.port = port;
7     }
8
9     public int getPort() {
10      return port;
11    }
12  }
```

```
throw new ServerTimedOutException("Imposible conectar", 80);
```



Aserciones

```
assert <expresión_booleana> ;
assert <expresión_booleana> ; <expresión_detallada> ;
```

- Si la expresión booleana no es true se provoca AssertionError y el programa aborta
- Chequean consistencia interna del programa
- Implementar programación por contrato:
  - Postcondiciones
  - Invariantes
  - Las precondiciones lanzan excepciones

Usos recomendados de las aserciones

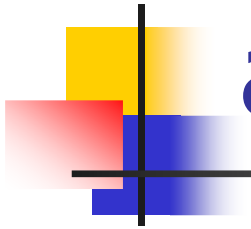
Postcondiciones e invariantes de clase

```
public Object pop() {
    int size = this.getElementCount();
    if (size == 0) {
        throw new RuntimeException("Intento de desapilar una pila vacía");
    }
    Object result = /* código para recuperar el elemento desapilado */;

    // probar la postcondición
    assert (this.getElementCount() == size - 1);
    return result;
}
```

Invariantes del flujo de control

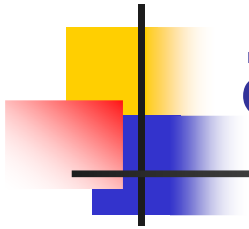
```
switch ( palo ) {
   case Palo.TREBOLES: // ...
       break;
   case Palo.DIAMANTES: // ...
       break;
   case Palo.CORAZONES: // ...
       break;
   case Palo.PICAS: // ...
       break;
   default: assert false : "Palo desconocido en la braja";
       break;
}
```

Usos inapropiados de las aserciones



- Verificación de los parámetros de un método public (precondiciones)
  - Se deben lanzar siempre Exception
    - IllegalArgumentException o
    - NullPointerException
- Las aserciones se deshabilitan cuando el programa está en producción, este chequeo no se tiene que deshabilitar

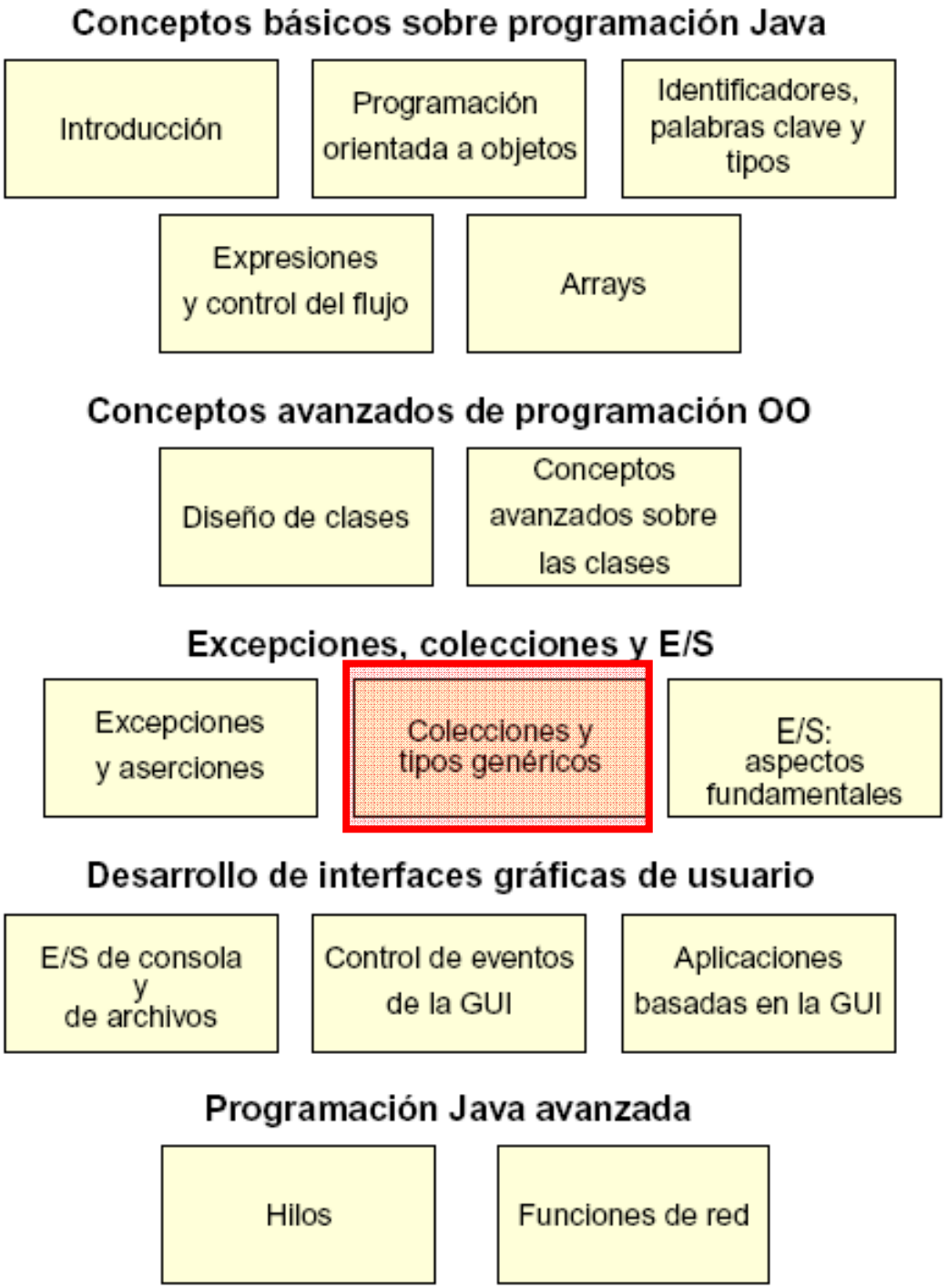
Control de la evaluación de las aserciones en runtime

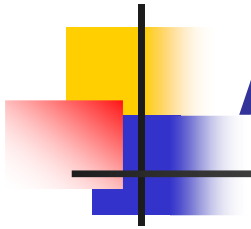


- Las aserciones son útiles durante el desarrollo y la fase de pruebas
- Una vez validado el programa, en explotación se deshabilitan para mejorar rendimiento
- Por defecto la JVM NO chequea asertos

```
java -enableassertions MiPrograma
java -ea MiPrograma
```

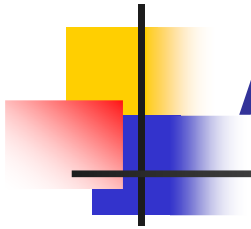
Contenidos


Conceptos básicos sobre programación Java
Introducción
Programación orientada a objetos
Identificadores, palabras clave y tipos
Expresiones y control del flujo
Arrays
Conceptos avanzados de programación OO
Diseño de clases
Conceptos avanzados sobre las clases
Excepciones, colecciones y E/S
Excepciones y aserciones
Colecciones y tipos genéricos
E/S: aspectos fundamentales
Desarrollo de interfaces gráficas de usuario
E/S de consola y de archivos
Control de eventos de la GUI
Aplicaciones basadas en la GUI
Programación Java avanzada
Hilos
Funciones de red



API Collections

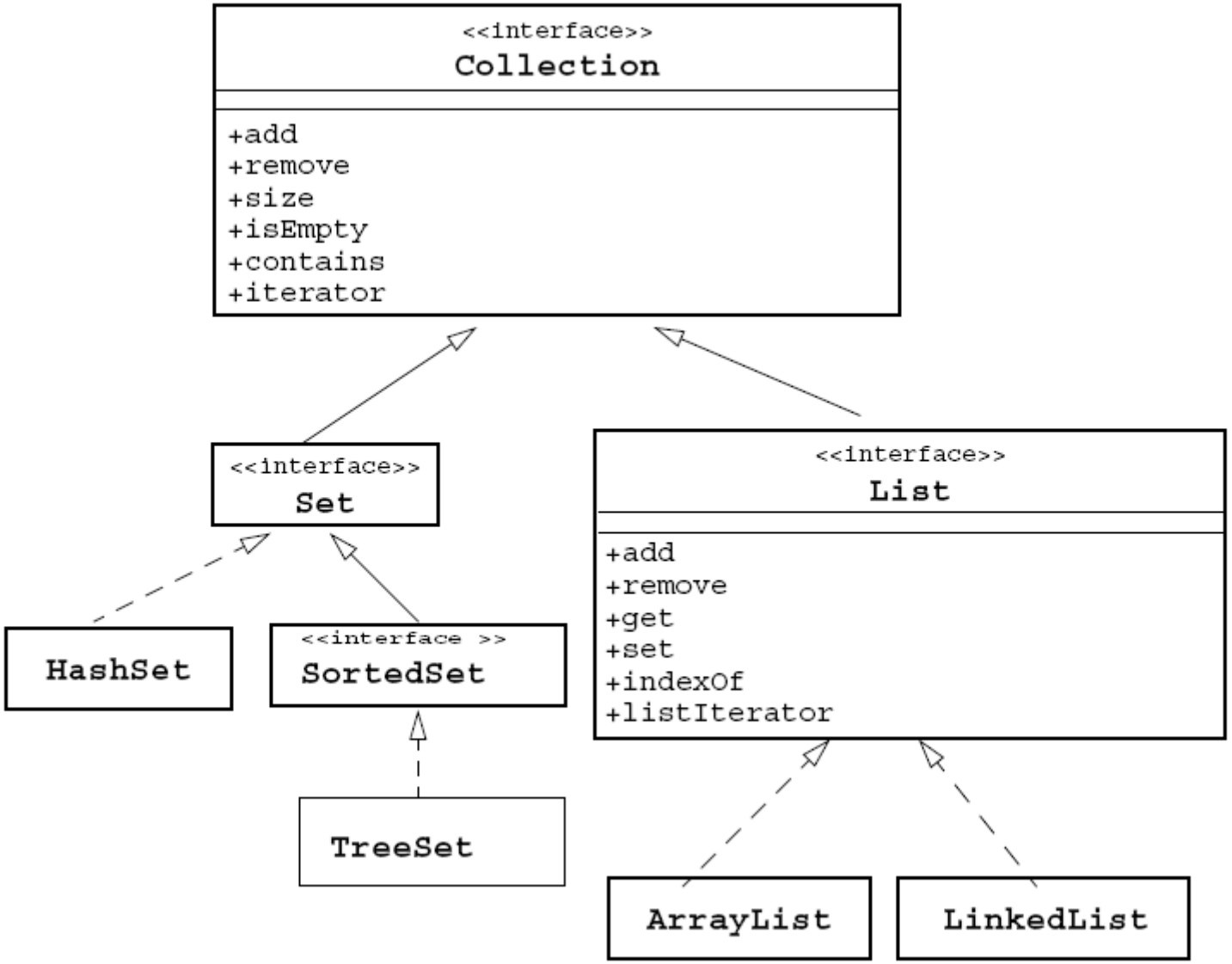
- Una *colección* es un objeto que administra un grupo de objetos
- Los objetos de la colección se llaman *elementos*
- Las colecciones suelen trabajar con numerosos tipos de objetos, pero todos ellos son de un mismo tipo (todos descienden de un mismo tipo de nivel superior)



API Collections

- El API Collections contiene interfaces que asignan los objetos a uno de los siguientes grupos:
- *Collection*: un grupo de objetos que se denominan elementos
  - *Set*: una colección sin orden específico, que no admite duplicados.
  - *List*: una colección ordenada que admite duplicados.

Jerarquía de Collections



Implementaciones de las interfaces

	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Deque`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

Ejemplo con Set

```
1    import java.util.*;
2    public class EjemploSet {
3      public static void main(String[] args) {
4       Set set = new HashSet();
5       set.add("one");
6       set.add("second");
7       set.add("3rd");
8       set.add(new Integer(4));
9       set.add(new Float(5.0F));
10      set.add("second");          // duplicado, no se agrega
11      set.add(new Integer(4)); // duplicado, no se agrega
12      System.out.println(set);
13     }
14   }
```

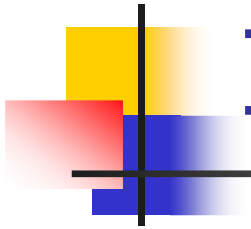
Salida → `[one, second, 5.0, 3rd, 4]`

Los elementos no aparecen en el mismo orden en el que se agregaron.

Ejemplo con List

```
1   import java.util.*;
2   public class EjemploList {
3     public static void main(String[] args) {
4     List list = new ArrayList();
5      list.add("one");
6      list.add("second");
7      list.add("3rd");
8      list.add(new Integer(4));
9      list.add(new Float(5.0F));
10     list.add("second");          // duplicado, se agrega
11     list.add(new Integer(4));    // duplicado, se agrega
12     System.out.println(list);
13    }
14  }
```

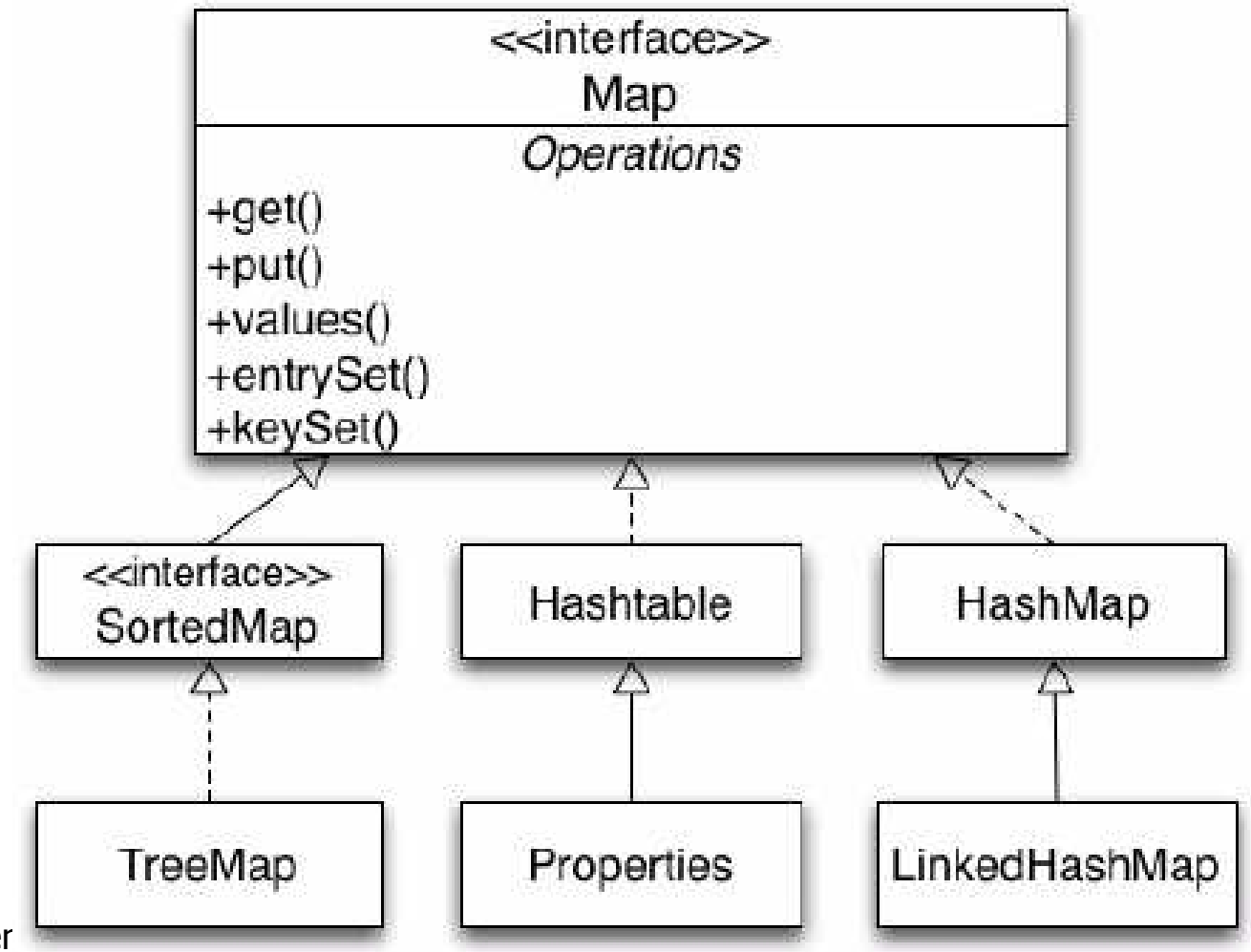
Salida → [one, second, 3rd, 4, 5.0, second, 4]



Interfaz Map

- También denominados arrays asociativos
- Un Map guarda asignaciones de claves a valores
- Map no admite claves duplicadas y una clave sólo puede asignarse a un valor
- Map: tres métodos para ver contenido como colecciones:
  - entrySet: devuelve una variable Set que contiene todos los pares formados por una clave y un valor.
  - keySet: devuelve una variable Set con todas las claves del mapa.
  - values: devuelve una variable Collection con todos los valores contenidos en el mapa.

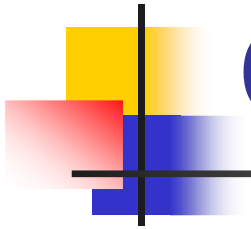
Jerarquía Map



Ejemplo con Map

```
[second, one, third]
[2, 1st, III]
[second=2, one=1st, third=III]
```

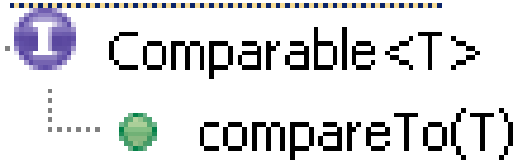
```
1   import java.util.*;
2   public class EjemploMap {
3      public static void main(String args[]) {
4         Map map = new HashMap();
5         map.put("one","1st");
6         map.put("second", new Integer(2));
7         map.put("third","3rd");
8         // Sobrescribe la asignación anterior
9         map.put("third","III");
10        // Devuelve el conjunto de las claves
11        Set set1 = map.keySet();
12        // Devuelve la vista Collection de los valores
13        Collection collection = map.values();
14        // Devuelve el conjunto de las asignaciones de claves a valores
15        Set set2 = map.entrySet();
16         System.out.println(set1 + "\n" + collection + "\n" + set2);
17     }
18  }
```



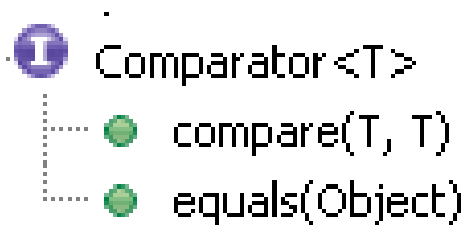
Ordenación de colecciones

- En las List los elementos se guardan en el orden de inserción
  - Las List se pueden ordenar creando una nueva lista con los elementos insertados en el orden adecuado → Collections.sort(...)
- En SortedSet y SortedMap se guardan siguiendo:
  - El orden "natural" de los elementos
  - O un orden especificado distinto del natural

Ordenación de collections



- Orden natural:
  - Las clase debe implementar el interfaz *Comparable*
- Otros criterios de ordenación:
  - Se deben implementar clases *Comparator*



Uso de comparable

```
import java.util.*;
public class ComparableTest {
    public static void main(String[] args) {
        Set estudianteSet = new TreeSet();
        estudianteSet.add(new Estudiante("Miguel", "Herraiz",101, 4,0));
        estudianteSet.add(new Estudiante("Juan", "Lino",102, 2,8));
        estudianteSet.add(new Estudiante("Jaime", "Marcos",103, 3,6));
        estudianteSet.add(new Estudiante("Clara", "Genio",104, 2,3));
        Object[] estudianteArray = estudianteSet.toArray();
        Estudiante s;
        for(Object obj : estudianteArray){
            s = (Estudiante) obj;
            System.out.printf("Nombre = %s %s ID = %d Media = %.1f\n",
               s.getNombre(), s.getApellido(), s.getId(), s.getMedia());
        }
    }
}
```

```
class Estudiante implements Comparable {
    ...
    double media = 0,0;

    public Estudiante(String Nombre, ...){
        ...
    }
    ...
    public double getMedia() { return media; }

    public int compareTo(Object o) {
        double f = media-((Estudiante)o).getMedia();
        if (f == 0,0)
            return 0; // 0 --> son iguales
        else if (f<0,0)
            return -1; // < 0 --> menos que o antes de
        else
            return 1; // > 0 --> más que o después de
    }
}
```

```
Nombre = Clara Genio ID = 104 Media = 2,3
Nombre = Juan Lino ID = 102 Media = 2,8
Nombre = Jaime Marcos ID = 103 Media = 3,6
Nombre = Miguel Herraiz ID = 101 Media = 4,0
```

Uso de Comparator

```
class Estudiante {
   Long id;
   String nombre;
   String apellidos;
   double media = 0,0;

   public Estudiante(String Nombre, ...){
      ...
   }
   ...
   public double getId() { return id; }
   public double getNombre() { return nombre; }
   public double getApellidos { return apellidos; }
   public double getMedia() { return media; }
}
```

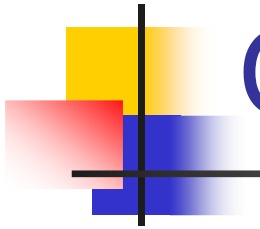
```
public class CompNombre implements Comparator {
   public int compare(Object o1, Object o2) {
      Estudiante e1 = (Estudiante) o1;
      Estudiante e2 = (Estudiante) o2;
      return e1.getNombre().compareTo(e2.getNombre());
   }
}
```

```
public class CompMedia implements Comparator {
   public int compare(Object o1, Object o2) {
      Estudiante e1 = (Estudiante) o1;
      Estudiante e2 = (Estudiante) o2;
      return e1.getMedia() - e1.getMedia();
   }
}
```

Uso de Comparator

```
import java.util.*;
public class ComparableTest {
    public static void main(String[] args) {
        Set estudianteSet = new TreeSet(new CompNombre);
        estudianteSet.add(new Estudiante("Miguel", "Herraiz",101, 4,0));
        estudianteSet.add(new Estudiante("Juan", "Lino",102, 2,8));
        estudianteSet.add(new Estudiante("Jaime", "Marcos",103, 3,6));
        estudianteSet.add(new Estudiante("Clara", "Genio",104, 2,3));
        Object[] estudianteArray = estudianteSet.toArray();
        Estudiante s;
        for(Object obj : estudianteArray){
            s = (Estudiante) obj;
            System.out.printf("Nombre = %s %s ID = %d Media = %.1f\n",
               s.getNombre(), s.getApellido(), s.getId(), s.getMedia());
        }
    }
}
```

```
Nombre = Jaime Marcos ID = 103 Media = 3,6
Nombre = Juan Lino ID = 102 Media = 2,8
Nombre = Clara Genio ID = 104 Media = 2,3
Nombre = Miguel Herraiz ID = 101 Media = 4,0
```



Colecciones con Genéricos

- Sin genéricos

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

- Con genéricos

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

- Con genéricos y autoboxing

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

Ejemplo con Set genérico

```
1   import java.util.*;
2   public class EjemploSetGen {
3    public static void main(String[] args) {
4      Set<String> set = new HashSet<String>();
5      set.add("one");
6      set.add("second");
7      set.add("3rd");
8      // Esta línea genera un error de compilación
9      set.add(new Integer(4));
10     // Duplicado, no se agrega
11     set.add("second");
12     System.out.println(set);
13   }
14  }
```

Comparación

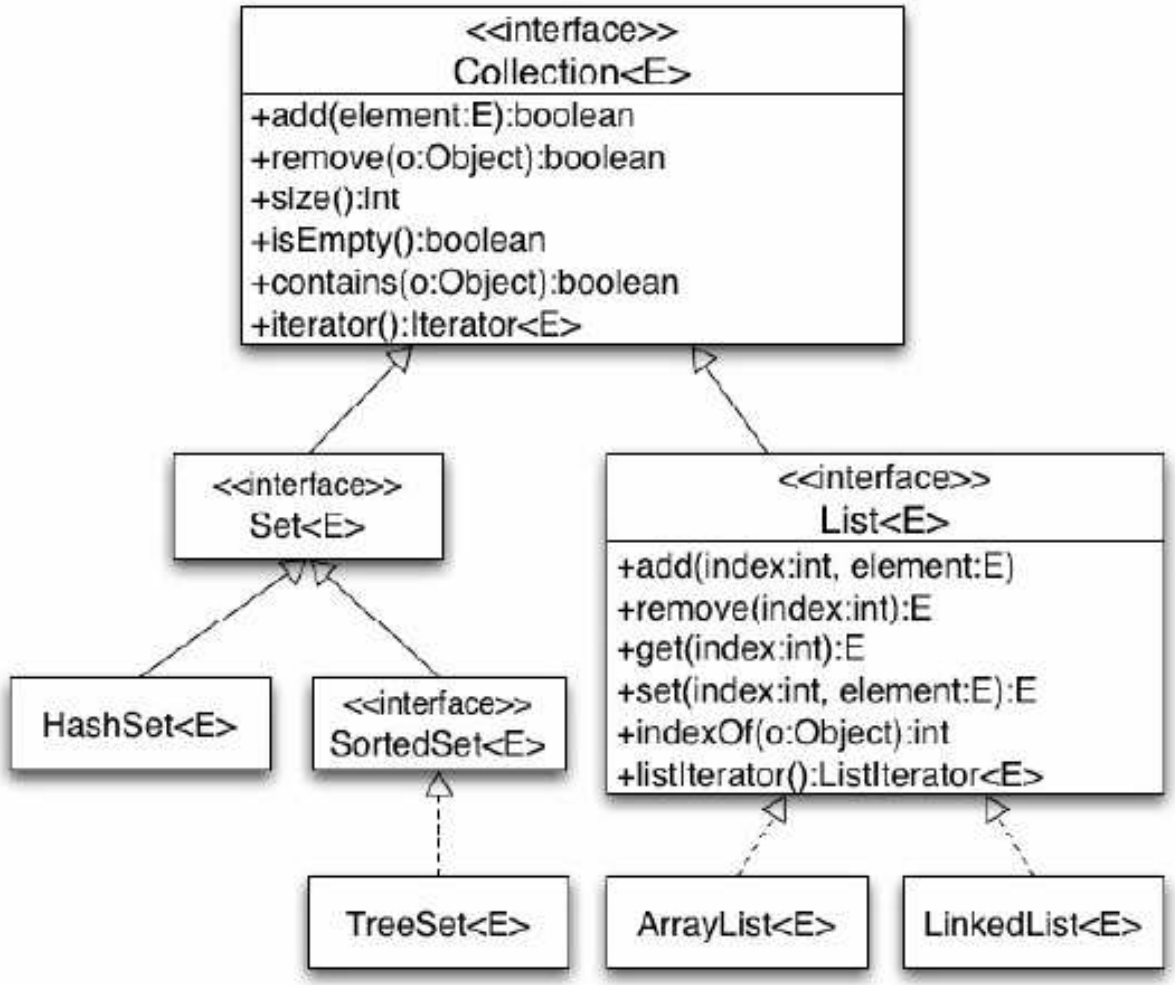
Categoría	Clase no genérica	Clase genérica
Declaración de clase	`public class ArrayList extends AbstractList implements List`	`public class ArrayList<E> extends AbstractList<E> implements List <E>`
Declaración de constructor	`public ArrayList (int capacity)`	`public ArrayList (int capacity)`

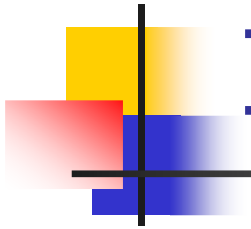
Declaración de método	`public void add(Object o)`
	
	`public Object get(int index)`
		`public void add(E o)`
		
		`public E get(int index)`

Ejemplos de declaraciones de variables	`ArrayList list1;`
	
	`ArrayList list2;`
		`ArrayList <String> a3;`
		
		`ArrayList <Date> a4;`

Ejemplos de declaraciones de instancias	`list1 = new ArrayList(10);`
	
	`list2 = new ArrayList(10);`
		`a3= new ArrayList<String> (10);`
		
		`a4= new ArrayList<Date> (10);`

API Collections ahora genérico





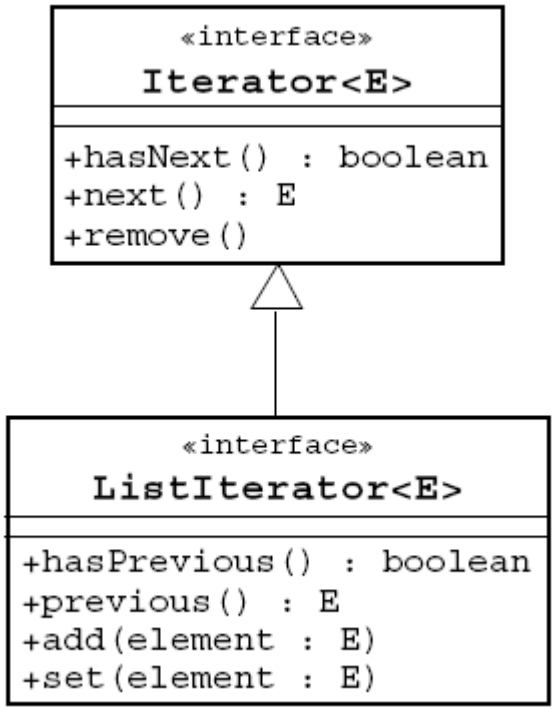
Iteradores

- Permiten recorrer la colección hacia delante

```
1   List list<Estudiante> = new ArrayList<Estudiante>();
2   // agregue algunos elementos
3   Iterator<Estudiante> elements = list.iterator();
4   while (elements.hasNext()) {
5     System.out.println(elements.next());
6   }
```

- El orden en el que se encuentran es determinista o no dependiendo del tipo de colección
  - Set no es determinista
  - SortedSet si lo es, orden natural
  - List tb lo es, orden de inserción

Iteradores



- Todas las colecciones producen iteradores
- List soporta un iterador especial con funcionalidad añadida
  - Recorrido inverso
  - Inserción
  - Modificación

Bucle for mejorado

```
1   public void deleteAll(Collection<NameList> c) {
2     for (Iterator<NameList> i=c.iterator(); i.hasNext();)
{
3       NameList nl = i.next();
4       nl.deleteItem();
5     }
6   }
```

```
1   public void deleteAll(Collection<NameList> c) {
2     for (NameList nl: c) {
3       nl.deleteItem();
4     }
5   }
```

```
1   List<Asignatura> asignaturas=...;
2   List<Profesor> profesores=...;
3   List<Curso> listaCursos = new ArrayList<Curso>();
4   for (Asignatura asig: asignaturas) {
5     for (Profesor prof: profesores) {
6       listaCursos.add(new Curso(asig, prof));
7     }
8   }
```

Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

Desarrollo de interfaces gráficas de usuario

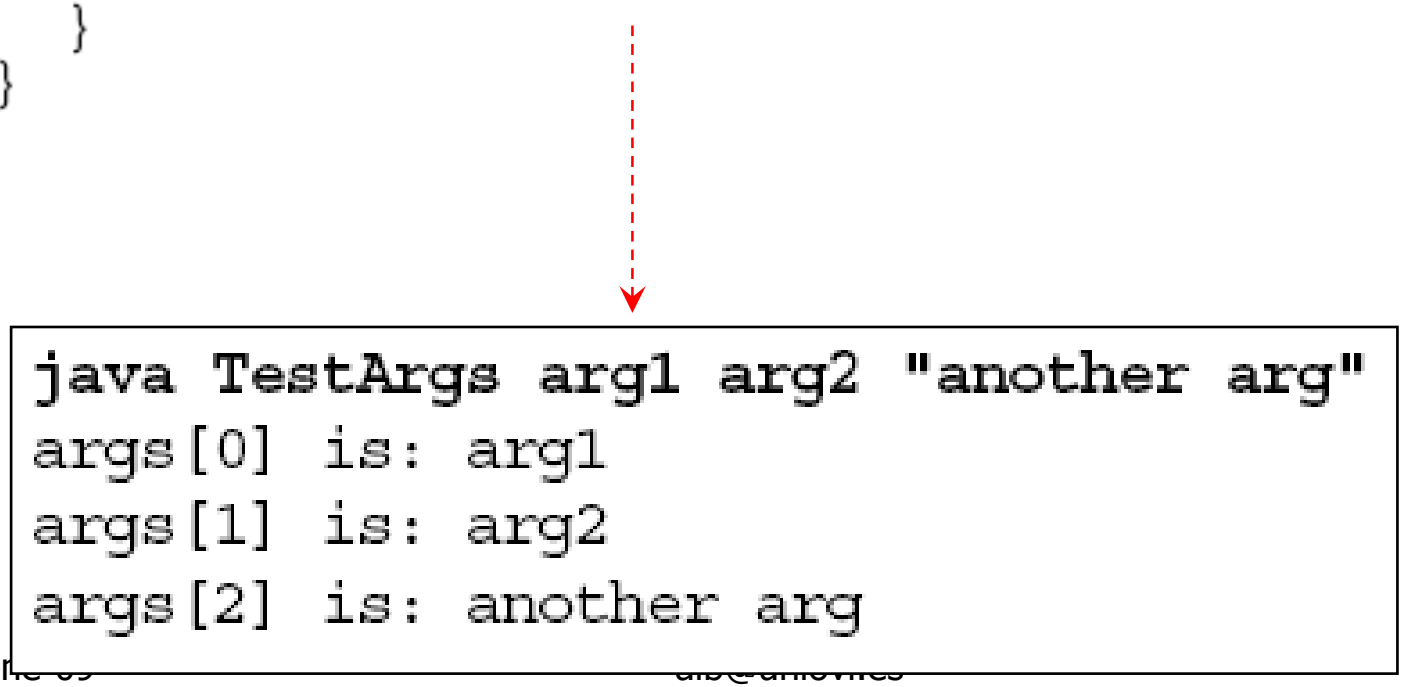
- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

Programación Java avanzada

- Hilos
- Funciones de red

Argumentos de la línea de comandos

```
1   public class TestArgs {
2      public static void main(String[] args) {
3         for (int i = 0; i < args.length; i++) {
4            System.out.println("args[" + i + "] is: " + args[i]);
5         }
6      }
7   }
```



Propiedades del sistema

```
1   import java.util.Properties;
2
3   public class TestProperties {
4      public static void main(String[] args) {
5         Properties props = System.getProperties();
6         props.list(System.out);
7      }
8   }
```

```
java -DmyProp=theValue TestProperties
```

```
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\jse\jdk1.6.0\jre\bin
java.vm.version=1.6.0-b105
java.vm.vendor=Sun Microsystems Inc.
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
user.country=US
myProp=theValue
```

- Properties almacena pares clave=valor, ambos String
- Estas propiedades son las que "conoce" la JVM en el momento del arranque

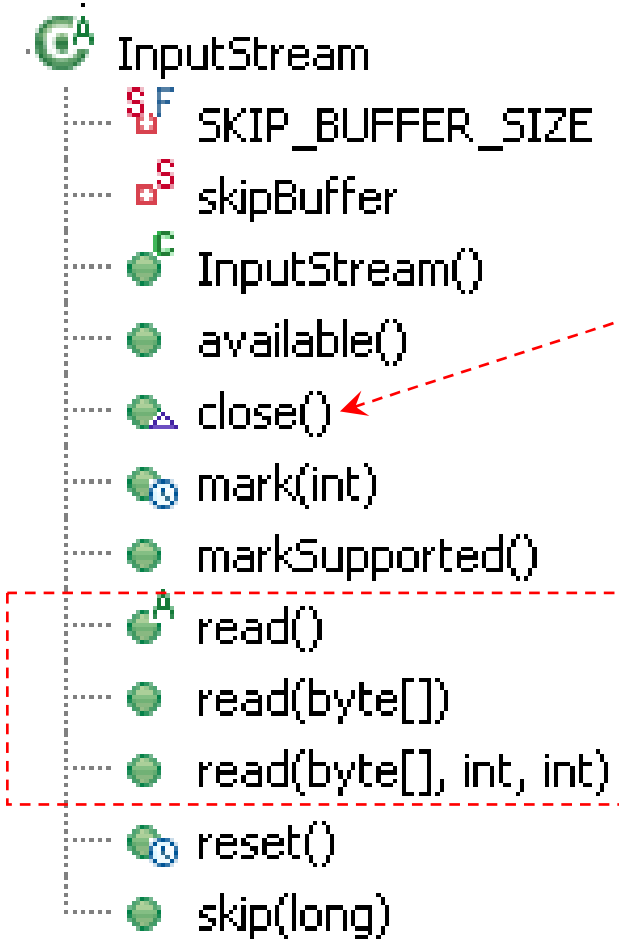
Principios básicos del flujo de E/S

- Un *flujo* es una secuencia de datos procedentes de una *fuente* en dirección a un *sumidero*
  - Normalmente, un programa representa un extremo de ese flujo, mientras que el otro extremo lo constituye otro nodo (por ejemplo un archivo)
- Las fuentes y sumideros también se denominan *flujos de entrada* y *flujos de salida*
  - Un flujo de entrada solo se lee
  - Un flujo de salida solo se escribe

Clases fundamentales de flujos

Flujo	Flujos de bytes	Flujos de caracteres
Flujos fuente	`InputStream`	`Reader`
Flujos sumidero	`OutputStream`	`Writer`

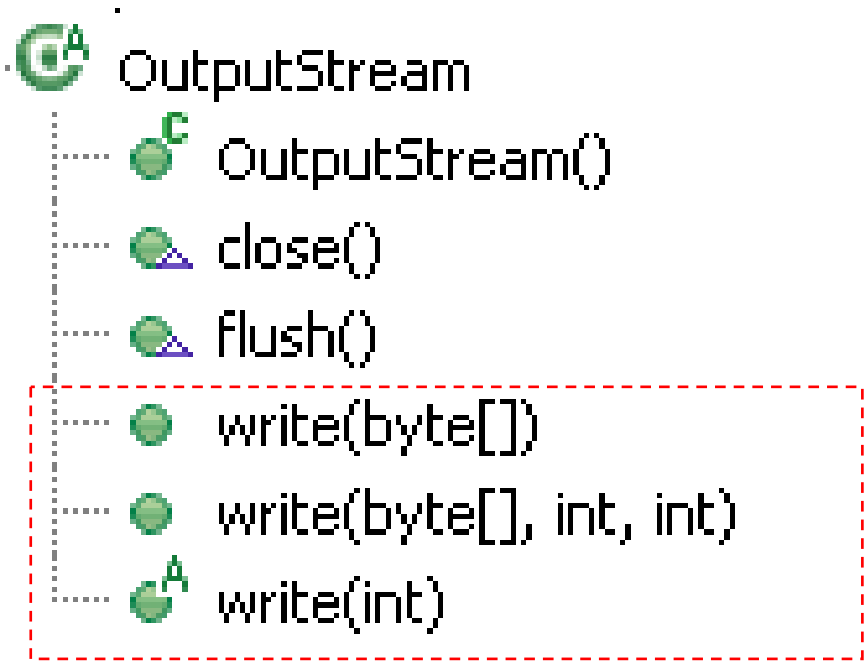
Flujos de bytes. Métodos InputStream



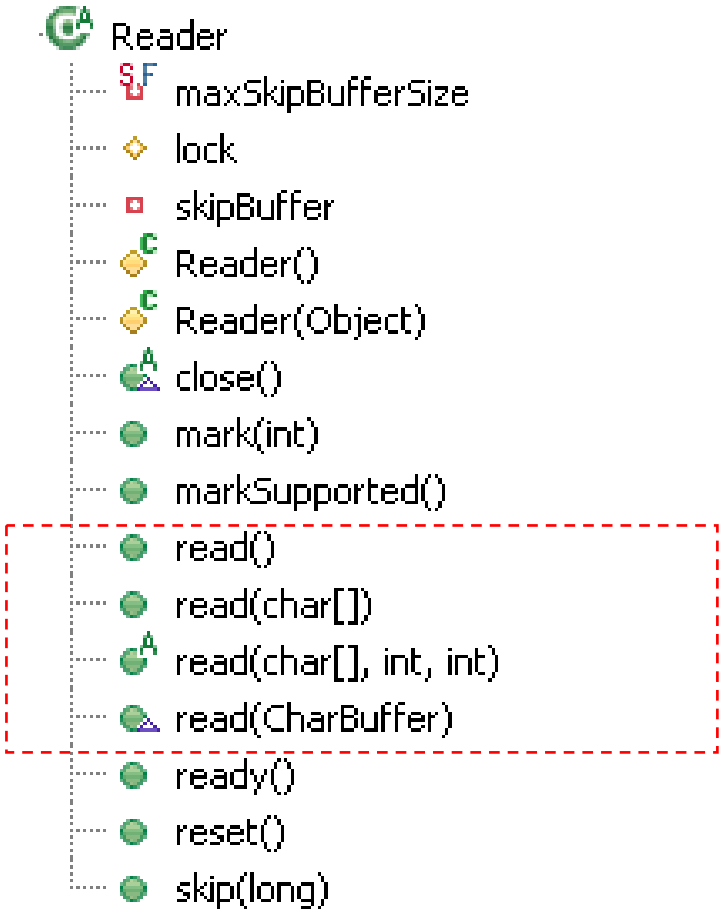
Siempre se deben cerrar al terminar
Hacerlo en un finally

Devuelven int que representan al byte leído

Flujos de bytes.
Métodos OutputStream



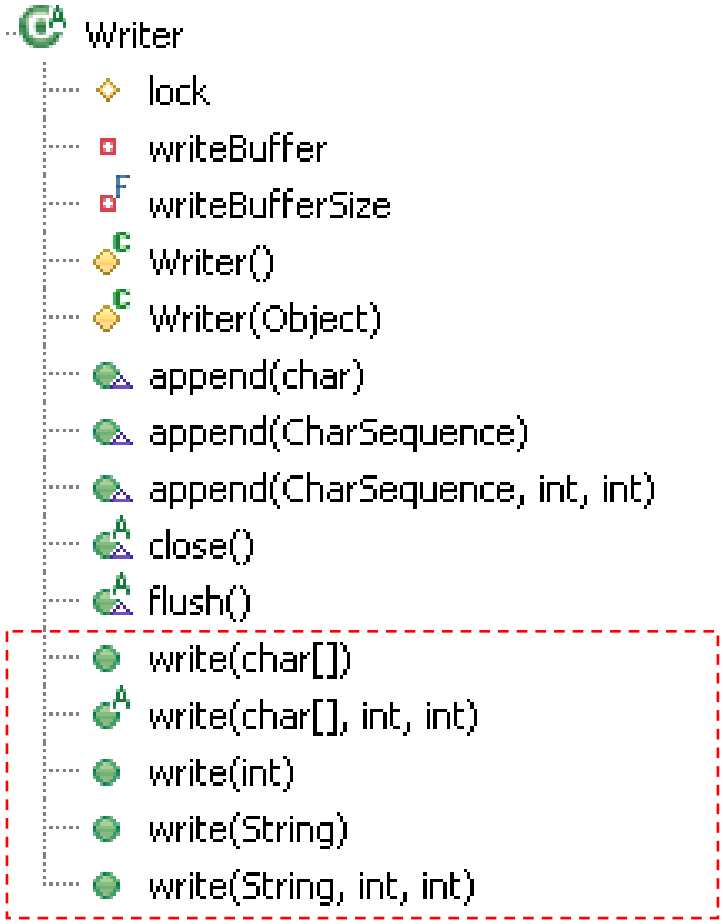
Flujos de caracteres. Métodos Reader




```
public int read() throws IOException {
```

Devuelve int representando
al carácter Unicode (32 bits) leído

Flujos de caracteres. Métodos Writer



Flujos de nodos

- Son los extremos finales de cadenas de flujos de procesamiento
  - Fuentes o sumideros

Tipo	Flujos de caracteres	Flujos de bytes
Archivo	`FileReader` `FileWriter`	`FileInputStream` `FileOutputStream`
Memoria: array	`CharArrayReader` `CharArrayWriter`	`ByteArrayInputStream` `ByteArrayOutputStream`
Memoria: cadena	`StringReader` `StringWriter`	N/D
Tubería	`PipedReader` `PipedWriter`	`PipedInputStream` `PipedOutputStream`

Reader y Writer: Ejemplo

```
public class PruebaFlujosNodos {
    public static void main(String[] args) {
        try {
            FileReader input = new FileReader(args[0]);
            try {
                FileWriter output = new FileWriter(args[1]);
                try {
                    char[] buffer = new char[128];
                    int charsRead;

                    charsRead = input.read(buffer);
                    while ( charsRead != -1 ) {
                        output.write(buffer, 0, charsRead);
                        charsRead = input.read(buffer);
                    }
                } finally {
                    output.close();}
            } finally {
                input.close();}
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

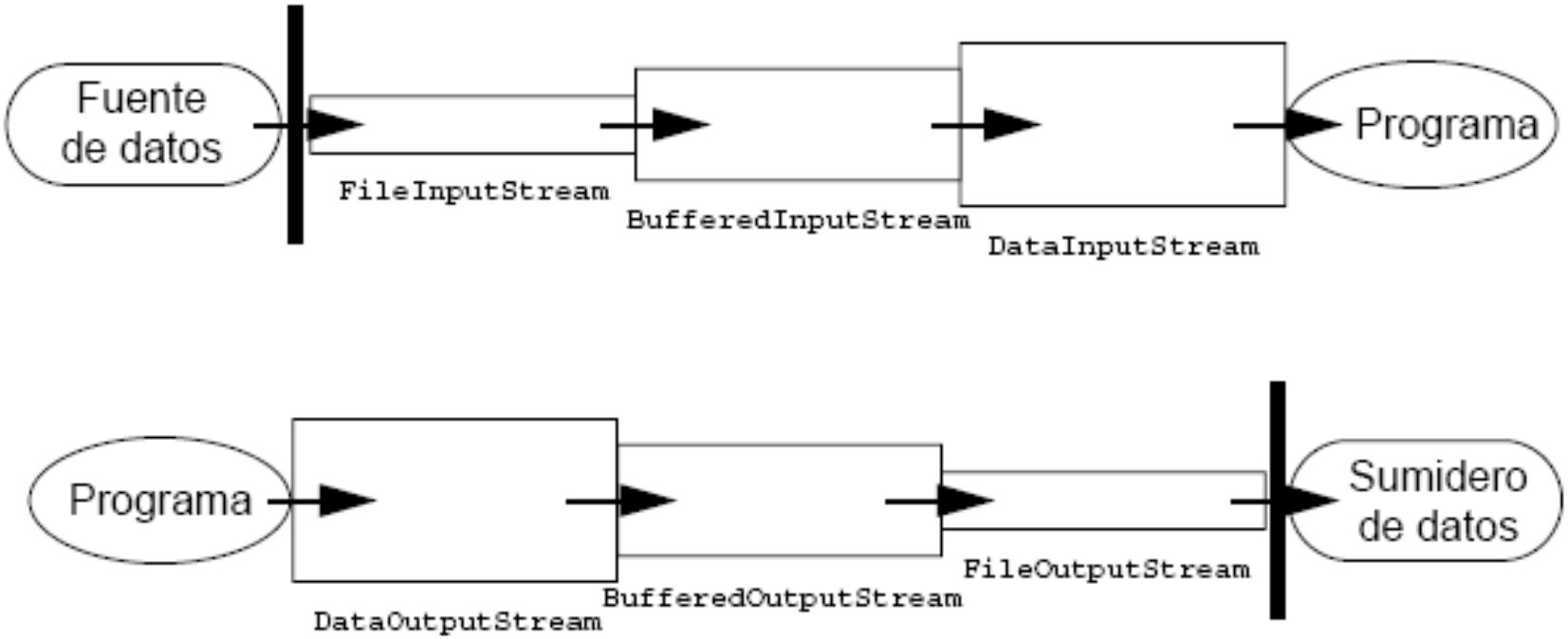
```
java PruebaFlujosNodos archivo1 archivo2
```

Flujos en búfer

```
public class PruebaFlujosBufer {
    public static void main(String[] args) {
        try {
            BufferedReader bufInput = new BufferedReader(new FileReader(args[0]));
            try {
                BufferedWriter bufOutput= new BufferedWriter(new FileWriter(args[1]));
                try {
                    String line = bufInput.readLine();
                    while ( line != null ) {
                        bufOutput.write(line);
                        bufOutput.newLine();
                        line = bufInput.readLine();
                    }
                } finally {
                    bufOutput.close();
                }
            } finally {
                bufInput.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Buffered[R/W] permite leer líneas enteras
- Añaden procesamiento a los char servidos por FileReader y FileWriter
- Son nodos intermedios de procesamiento

Concatenación de flujos de E/S



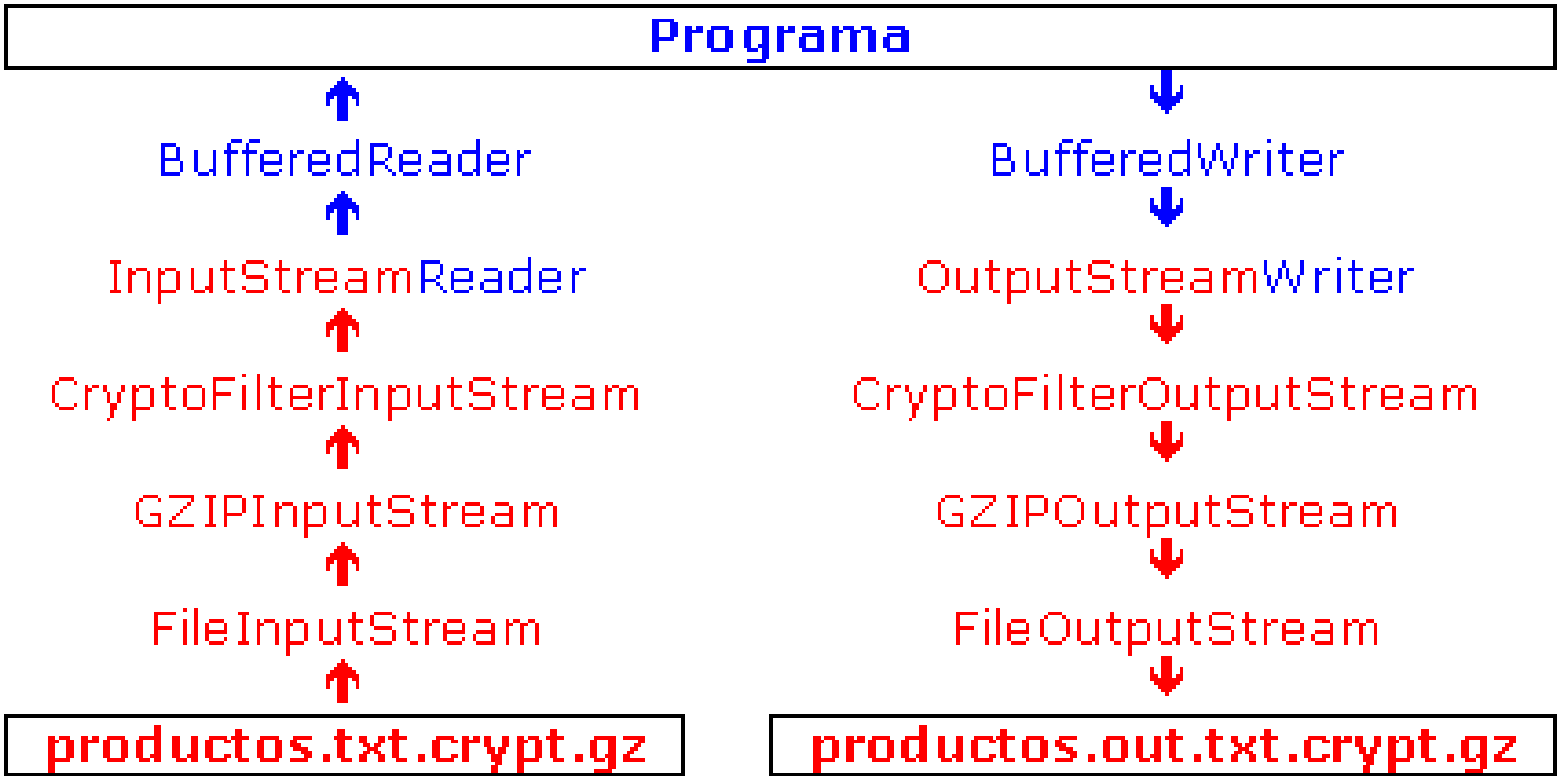
Flujos de procesamiento

Tipo	Flujos de caracteres	Flujos de bytes
Uso del búfer	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtrado	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Conversión entre bytes y caracteres	InputStreamReader OutputStreamWriter	
*Serialización de objetos		ObjectInputStream ObjectOutputStream
Conversión de datos		DataInputStream DataOutputStream
Recuento	LineNumberReader	LineNumberInputStream
Exploración	PushbackReader	PushbackInputStream
Impresión	PrintWriter	PrintStream

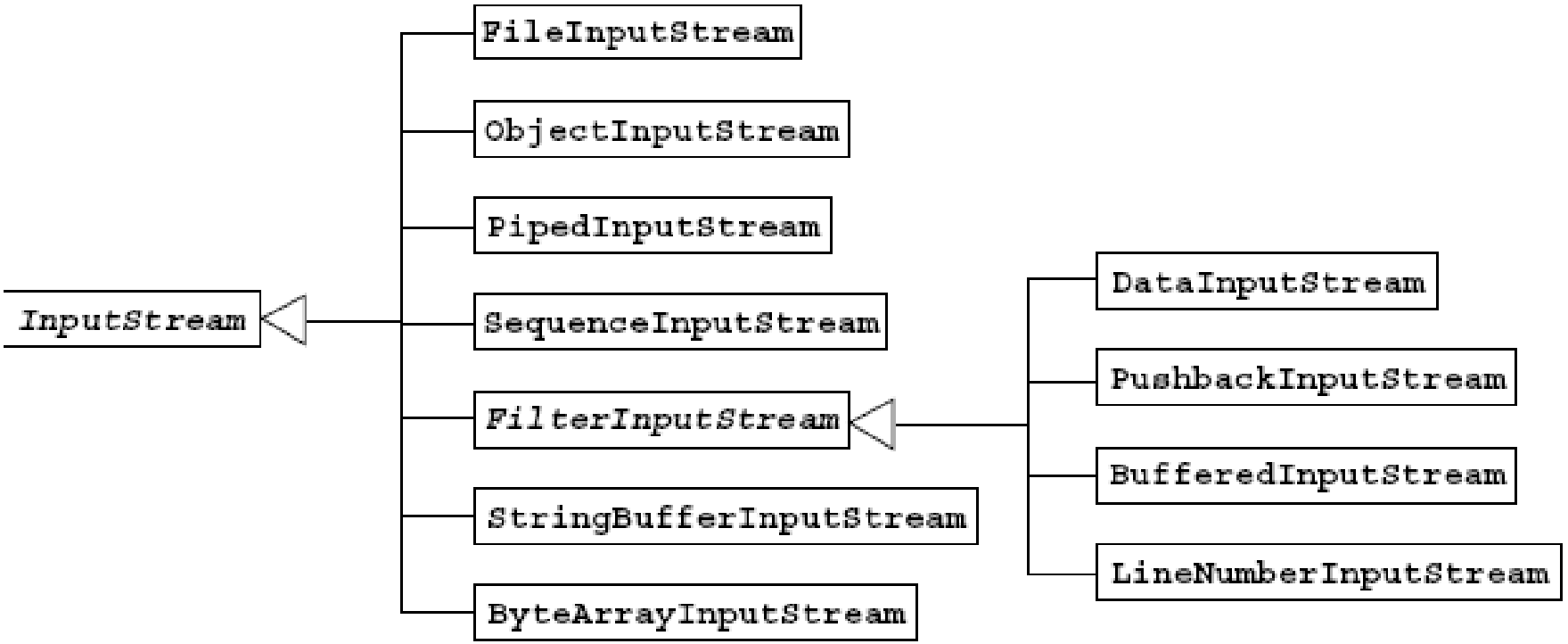
Clases de utilidad para implementar procesamiento particular

- Van en la parte intermedia de las cadenas de procesamiento
  - No son nodos extremos

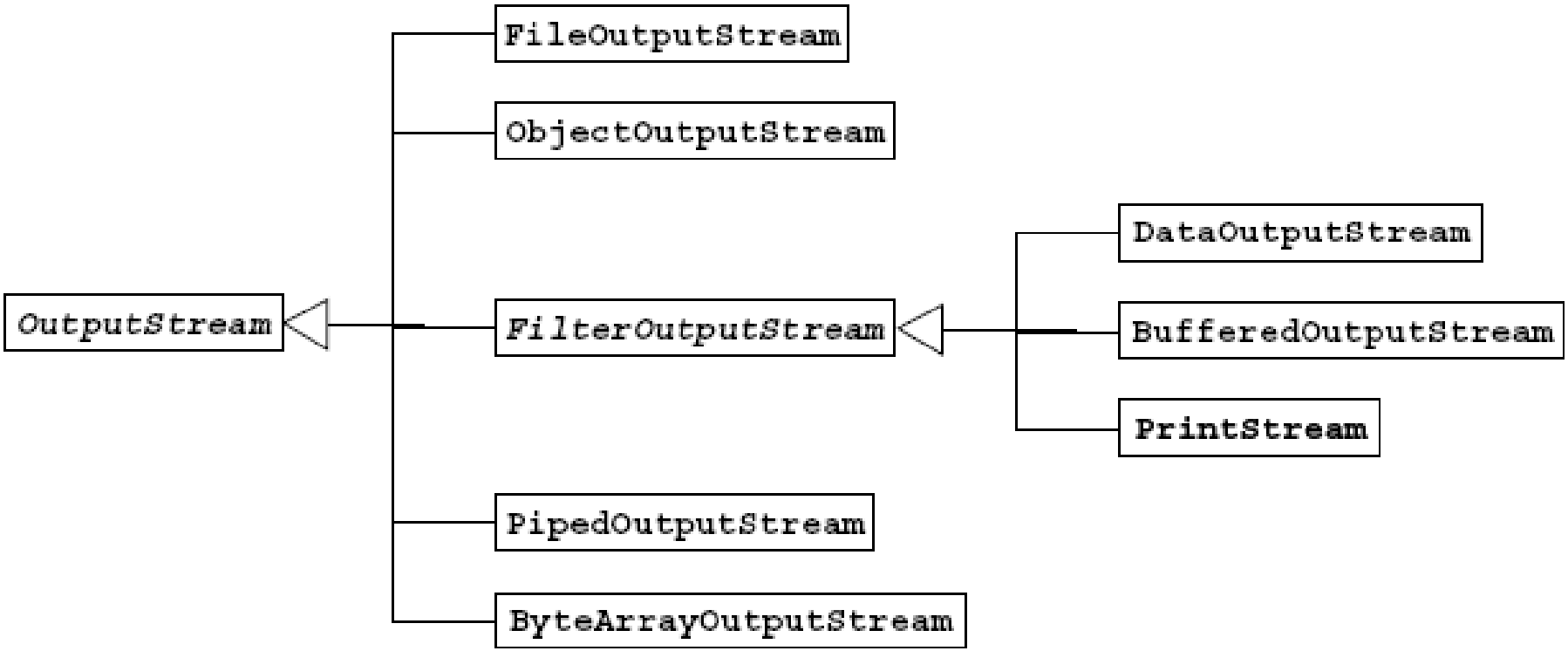
Ejemplo de cadenas de procesamiento



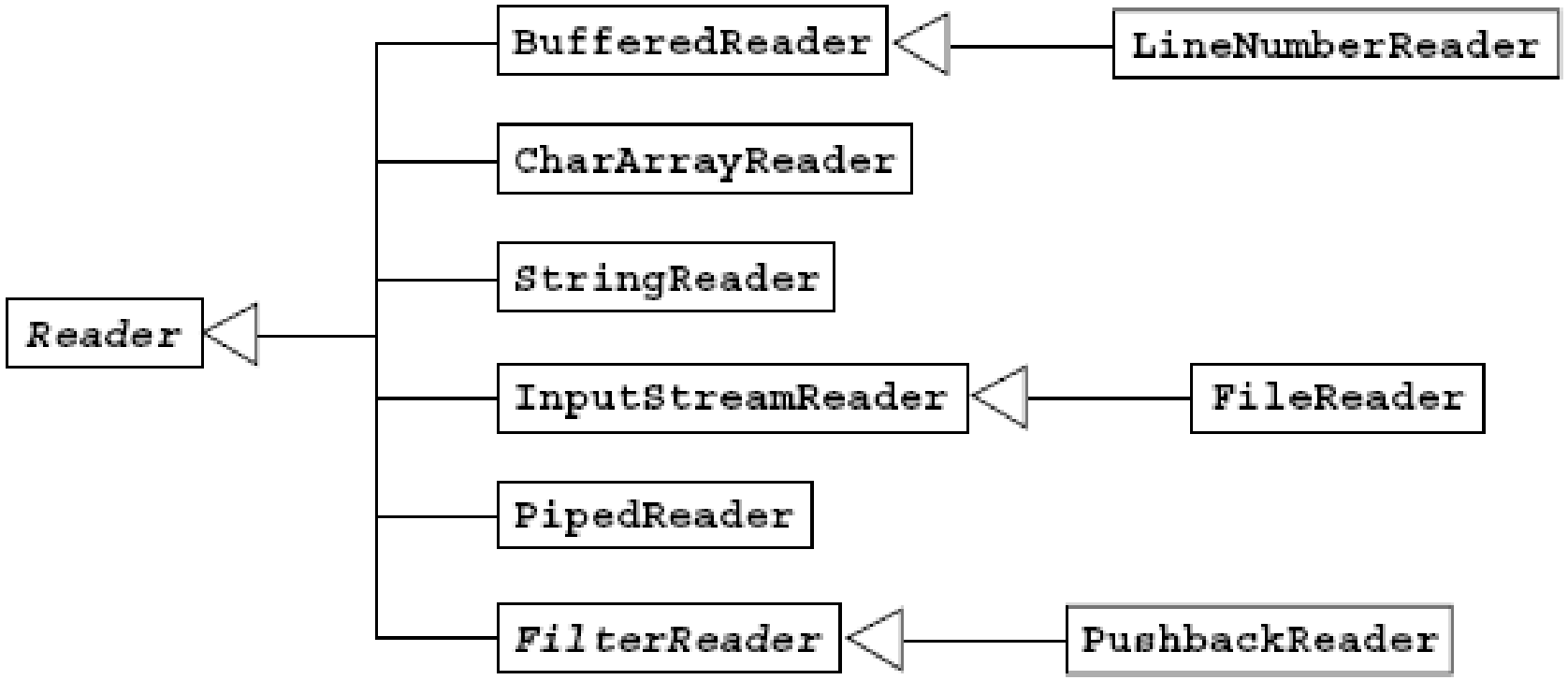
Clases básicas de flujos de bytes



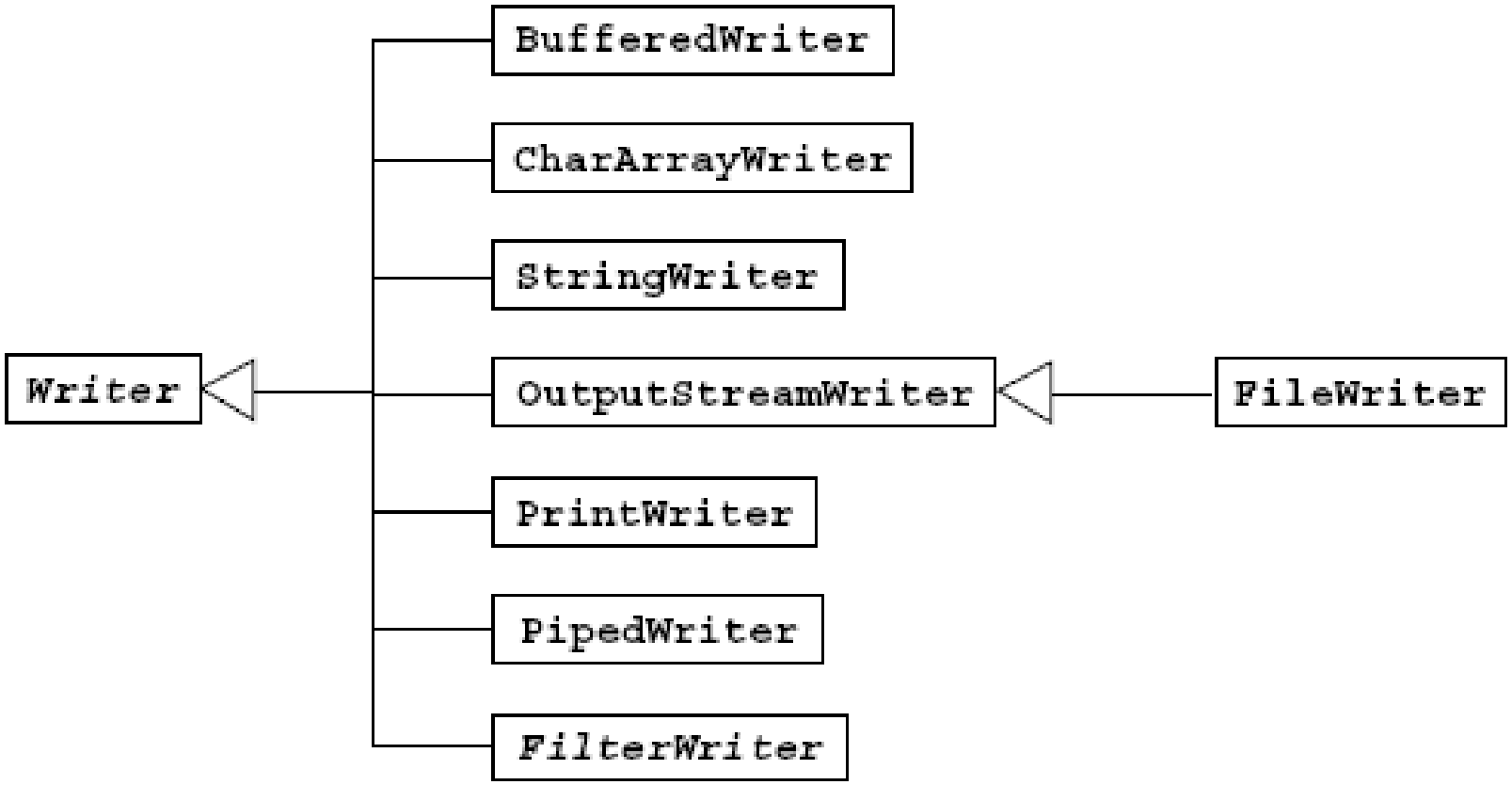
Clases básicas de flujos de bytes

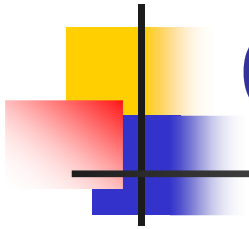


Clases básicas de flujos de caracteres



Clases básicas de flujos de caracteres



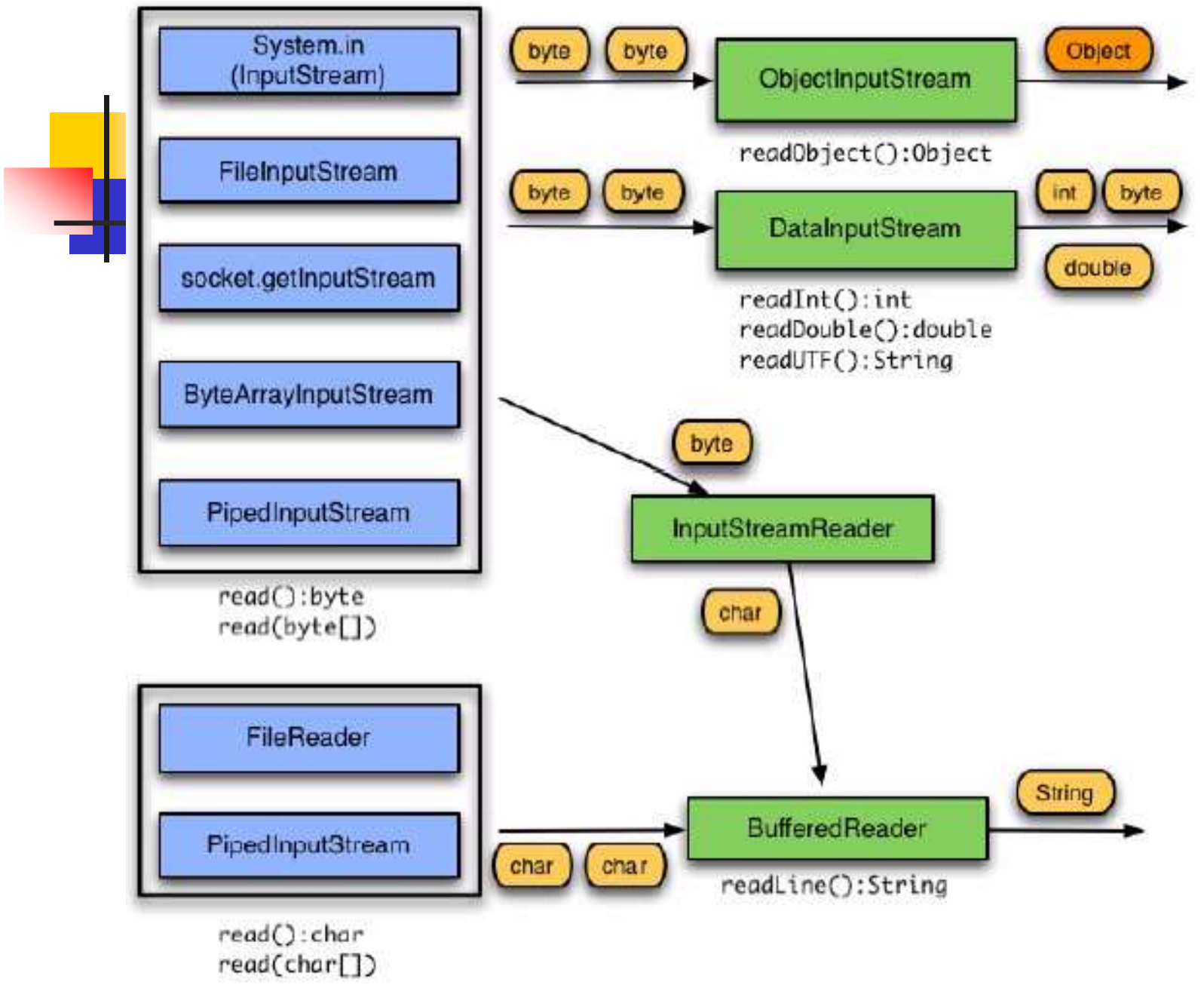


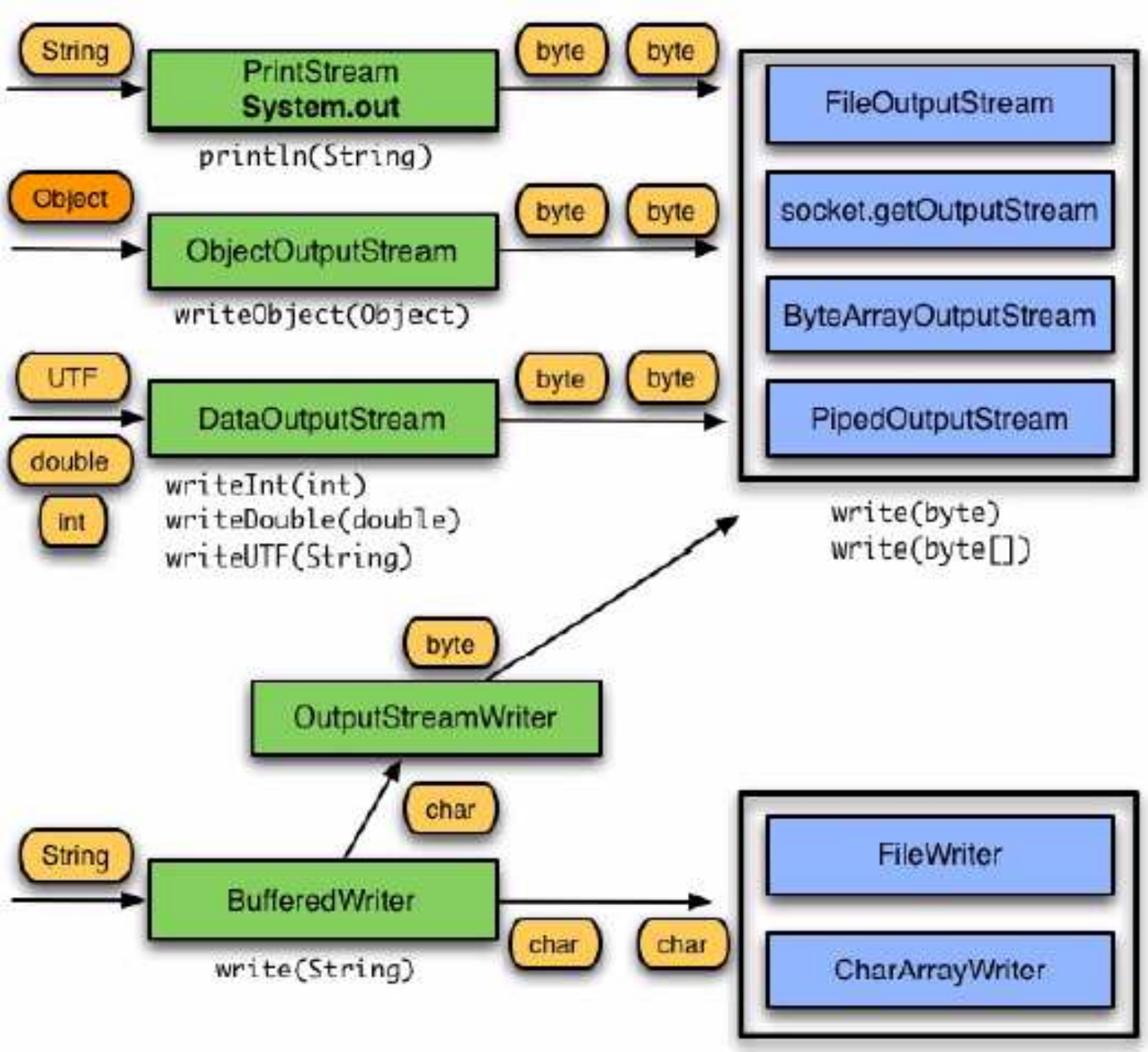
Codificaciones de caracteres

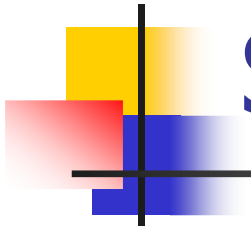
- UTF-8, UTF-16, ISO-8859-1, etc
  - Diferencia entre tabla de caracteres y representación binaria de los caracteres
  - Flujos de texto reconocen las codificaciones binarias

```
InputStreamReader ir
  = new InputStreamReader(System.in, "ISO-8859-1");
```

- Es posible indicar a los Flujos de caracteres el uso de Codificadores


System.in
(InputStream)
FileInputStream
socket.getInputStream
ByteArrayInputStream
PipedInputStream
read():byte
read(byte[])
byte
byte
ObjectInputStream
Object
readObject():Object
byte
byte
DataInputStream
int
byte
double
readInt():int
readDouble():double
readUTF():String
byte
InputStreamReader
char
FileReader
PipedInput3tream
read():char
read(char[])
char
char
BufferedReader
String
readLine():String


String
PrintStream
System.out
println(String)
byte
byte
Object
ObjectOutputStream
writeObject(Object)
byte
byte
UTF
double
int
DataOutputStream
writeInt(int)
writeDouble(double)
writeUTF(String)
byte
byte
FileOutputStream
socket.getOutputStream
ByteArrayOutputStream
PipedOutputStream
write(byte)
write(byte[])
byte
OutputStreamWriter
char
String
BufferedWriter
write(String)
char
char
FileWriter
CharArrayWriter



Serialización

- Convertir un objeto a una secuencia de bytes que puede:
  - Ser guardada en soporte persistente
  - Ser transmitida por la red
- Sólo se serializan los atributos, no los métodos
- La clase debe implementar el interfaz Serializable → no tiene métodos, solo marca
  - Todos los atributos deben ser serializables
  - Si alguno no lo puede ser se debe marcar como transient
- Se serializan grafos de objetos
  - Si un objeto referencia a otros (y estos a su vez…) se serializan todos los referenciados (a no ser que sean transient)

Clases serializables

```
1   public class MyClass implements Serializable {
2      public transient Thread myThread;
3        private String customerID;
4        private int total;
5   }
```

```
1   public class MyClass implements Serializable {
2      public transient Thread myThread;
3      private transient String customerID;
4      private int total;
5   }
```

- Determinadas clases por su naturaleza no son serializables: Streams, Threads, etc.
- El modificador de acceso no tiene efecto

Escritura y lectura de un flujo de objetos

```
public class SerializeDate {
    public static void main (String args[]) {
        Date d = new Date ();
        ObjectOutputStream s =  new ObjectOutputStream (
                new FileOutputStream ("date.ser"));
        try {
            s.writeObject (d);
        } catch (IOException e) {
            e.printStackTrace ();
        } finally {
            s.close();
        }
    }
}
```

```
public class DeserializeDate {
    public static void main (String args[]) {
        ObjectInputStream s =   new ObjectInputStream (
                new FileInputStream ("date.ser"));
        try {
            Date d = s.readObject();
        } catch (IOException e) {
            e.printStackTrace ();
        } finally {
            s.close();
        }

        System.out.println("Deserialized Date object from date.s
        System.out.println("Date: " + d);
    }
}
```

Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

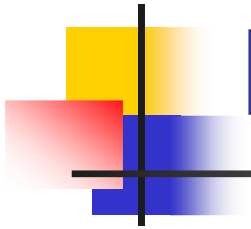
- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

Desarrollo de interfaces gráficas de usuario

- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

Programación Java avanzada

- Hilos
- Funciones de red

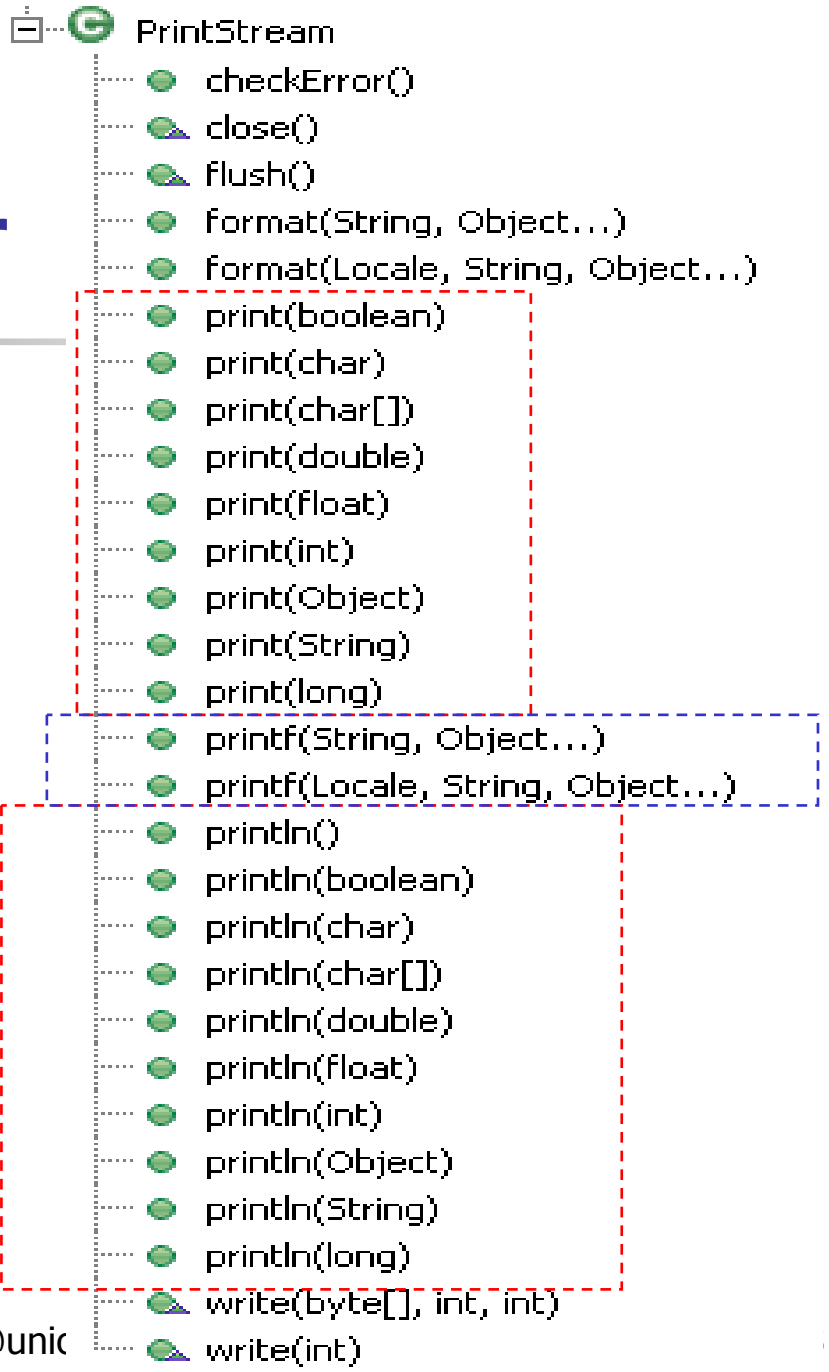


E/S de consola

- Java permite interacción con la consola usando java.lang.System
  - System.out: es un PrintStream inicialmente conectado a la ventana de terminal que haya abierto la aplicación Java
  - System.in: es un InputStream inicialmente conectado al teclado del usuario
  - System.err: es un PrintStream conectado también a la ventana de terminal

Escritura en salida estándar

```
void println(boolean)
void println(char)
void println(double)
void println(float)
void println(int)
void println(long)
void println(char[])
void println(Object)
```



Lectura de la entrada estándar

```
public class KeyboardInput {
    public static void main (String[] args) {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        System.out.println("Unix: Escriba ctrl-d para salir." +
            "\nWindows: Escriba ctrl-z para salir.");
        try {
          String s = in.readLine();
          while ( s != null ) {
              System.out.println("Leeido: " + s);
              s = in.readLine();
          }
        } catch (IOException e) {
          e.printStackTrace();
        } finally {
          in.close();
        }
    }
}
```

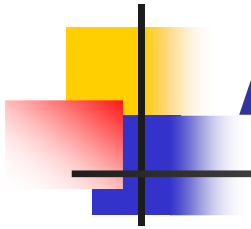
Salida en formato simple

```
System.out.printf("%s %5d %f%n",nombre,id,salario);

String s = String.format("%s %5d %f%n",nombre,id,salario);
System.out.print(s);
```

Código	Descripción
%s	Formatea el argumento como una cadena, normalmente llamando al método toString en el objeto.
%d %o %x	Formatea un entero como un valor decimal, octal o hexadecimal.
%f %g	Formatea un número de coma flotante. El código %g utiliza la notación científica.
%n	Introduce un carácter de salto de línea en la cadena o el flujo.
%%	Introduce el carácter % en la cadena o el flujo.

Entrada en formato simple

```
1   import java.io.*;
2   import java.util.Scanner;
3   public class ScanTest {
4      public static void main(String [] args) {
5         Scanner s = new Scanner(System.in);
6         String param = s.next();
7         System.out.println("parámetro 1" + param);
8         int value = s.nextInt();
9         System.out.println("segundo parámetro" + value);
10        s.close();
11     }
12  }
```

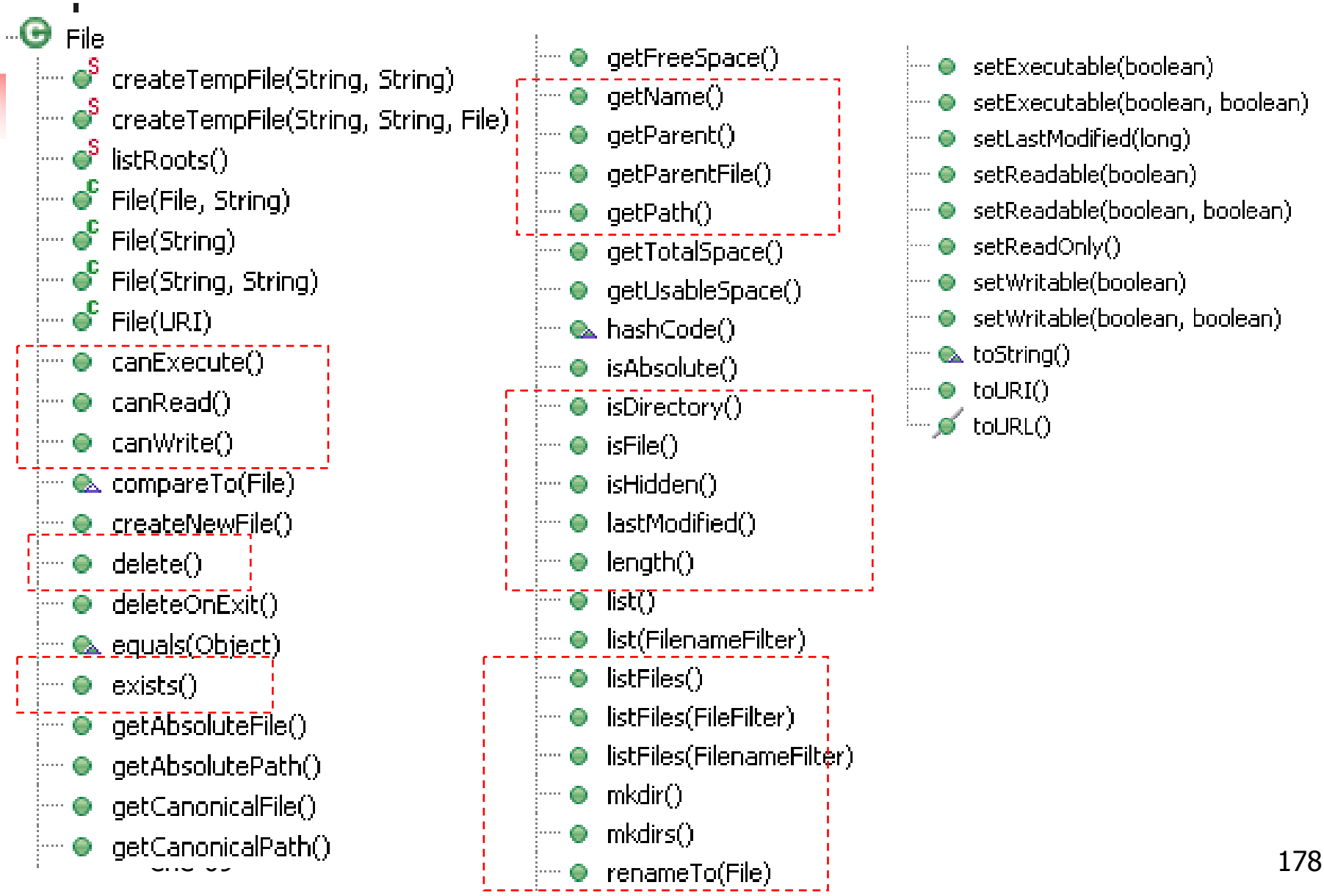


Archivos y E/S de archivos

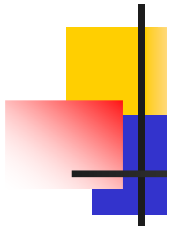
- Clase File
  - Proporciona utilidades para manejar información de ficheros y directorios de forma independiente de la plataforma
  - Pero no para leerlos ni escribirlos (para eso se usan flujos)

```
File miArchivo;
miArchivo = new File("miarchivo.txt");
miArchivo = new File("MisDocs", "miarchivo.txt");
File miDir = new File("MisDocs");
miArchivo = new File(miDir, "miarchivo.txt");
```

Clase File

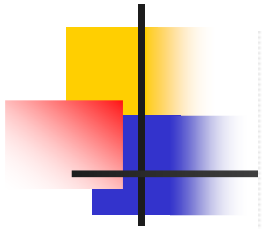

File
createTempFile(String, String)
createTempFile(String, String, File)
listRoots()
File(File, String)
File(String)
File(String, String)
File(URI)
canExecute()
canRead()
canWrite()
compareTo(File)
createNewFile()
delete()
deleteOnExit()
equals(Object)
exists()
getAbsoluteFile()
getAbsolutePath()
getCanonicalFile()
getCanonicalPath()
getFreeSpace()
getName()
getParent()
getParentFile()
getPath()
getTotalSpace()
getUsableSpace()
hashCode()
isAbsolute()
isDirectory()
isFile()
isHidden()
lastModified()
length()
list()
list(FilenameFilter)
listFiles()
listFiles(FileFilter)
listFiles(FilenameFilter)
mkdir()
mkdirs()
renameTo(File)
setExecutable(boolean)
setExecutable(boolean, boolean)
setLastModified(long)
setReadable(boolean)
setReadable(boolean, boolean)
setReadOnly()
setWritable(boolean)
setWritable(boolean, boolean)
toString()
toURI()
toURL()

Lectura de archivo caracteres



```
public class ReadFile {
    public static void main (String [] args) {
        File file = new File(args[0]);
        try {
            BufferedReader in = new BufferedReader(new FileReader(file));
            try {
                String s = in.readLine();
                while (s != null) {
                    System.out.println("Lectura: " + s);
                    s = in.readLine();
                }
            } finally {
                in.close();
            }
        } catch (FileNotFoundException e1) {
            System.err.println("Archivo no encontrado: " + file);
        } catch (IOException e2) {
            e2.printStackTrace();
        }
    }
}
```

Escritura de archivo caracteres



```
public class WriteFile {
    public static void main (String[] args) {
        File file = new File(args[0]);

        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));
            PrintWriter out = new PrintWriter(new FileWriter(file));
            System.out.print("Introduzca el texto del archivo. \n" +
                "[Escriba ctrl-d para terminar.]");

            String s;
            while ((s = in.readLine()) != null) {
                out.println(s);
            }
        } finally {
            in.close();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

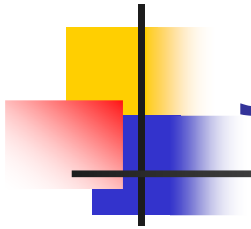
- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

Desarrollo de interfaces gráficas de usuario

- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

Programación Java avanzada

- Hilos
- Funciones de red



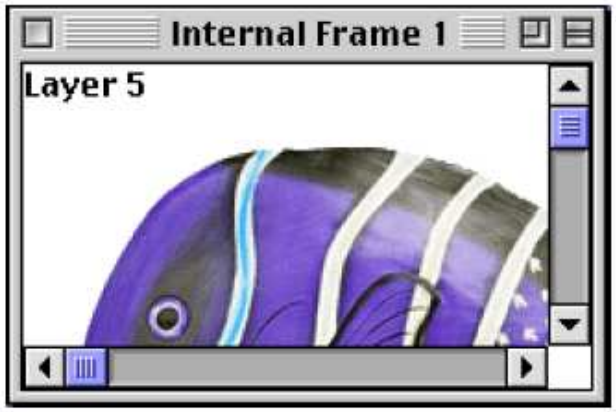
Java Foundation Classes (JFC)

- *JFC:* conjunto de paquetes para creación de interfaces gráficas (GUI)
  - Conjunto de componentes Swing (sustituyen a AWT)
  - Gráficos 2D:
    - dibujo avanzado, manipulaciones de color, formas y transformaciones (rotación, corte, deformación de objetos, etc.) y tratar texto
  - Estilo de interfaz adaptable (look-and-feel)
    - El mismo componente aparece con el aspecto del S.O. en el que se ejecuta
    - Windows, Motif y Metal.
  - API de accesibilidad.
    - lectores y amplificadores de pantallas, transformación de texto en voz, etc
  - Drag and Drop
  - Internacionalización

Look and feel

Metal

Mac OS

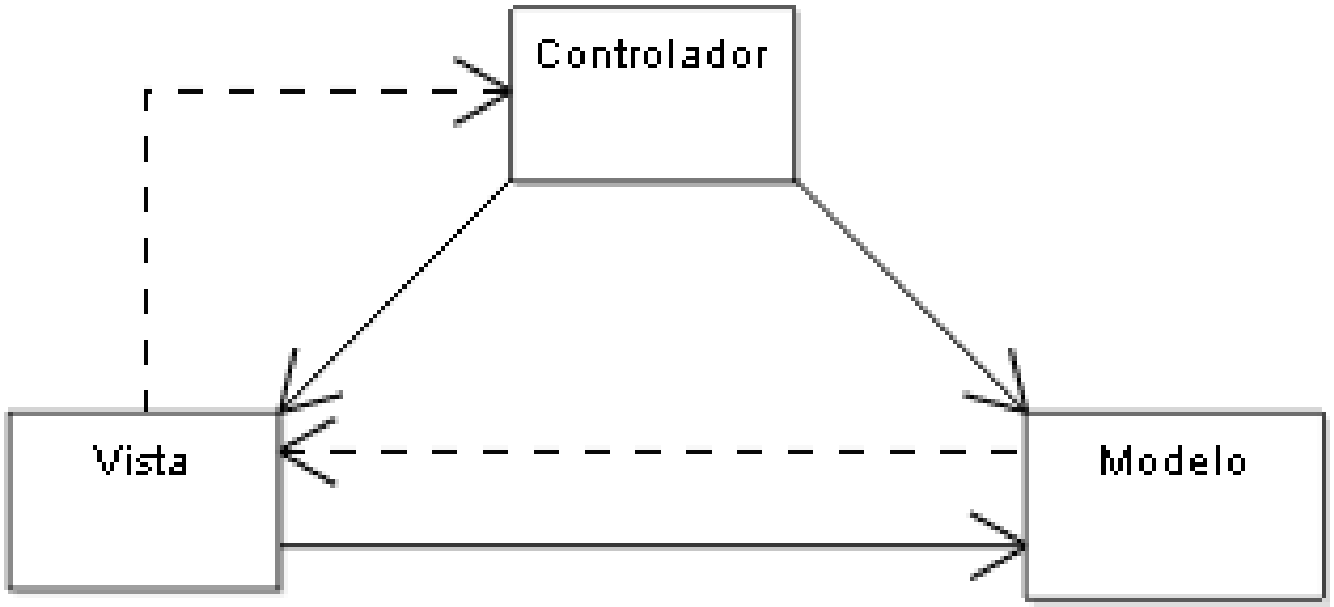


Motif

Windows

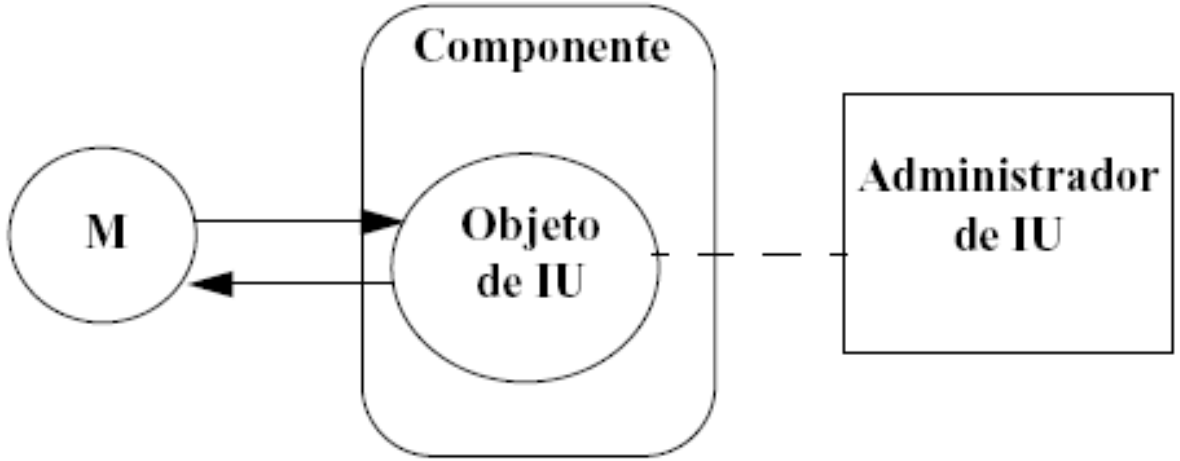
Arquitectura de Swing

- Se basa en el patrón MVC, pero con variantes

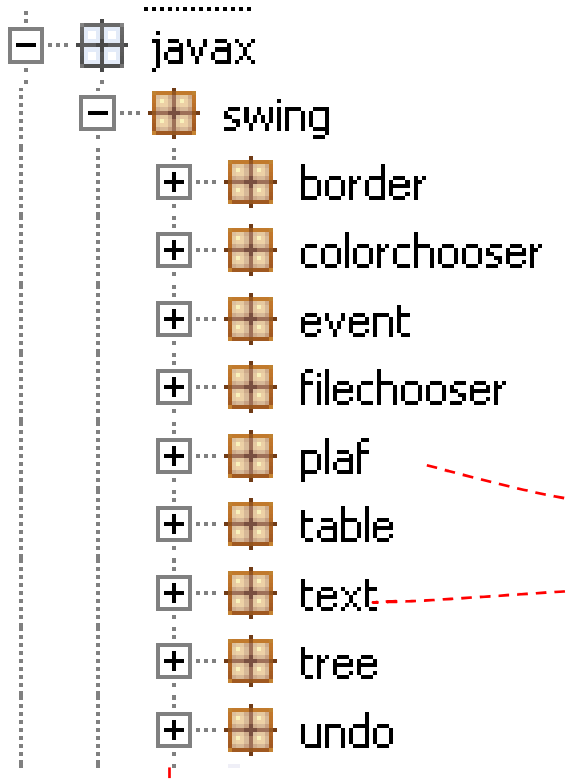


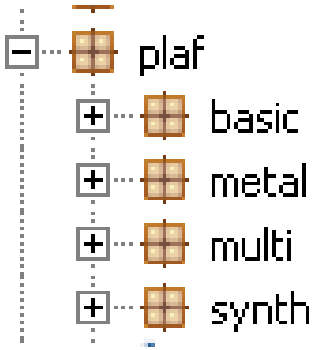
Arquitectura de modelo-delegado

- Variante de MVC usada por Swing
- La vista y el controlador se funden, solo queda separado el modelo
  - Es modelo suele ser una representación desde el punto de vista del GUI de la implementación real del modelo

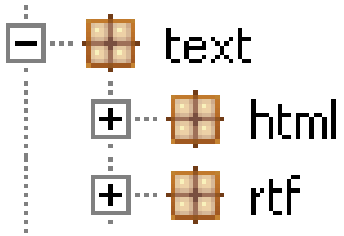


Paquetes de Swing





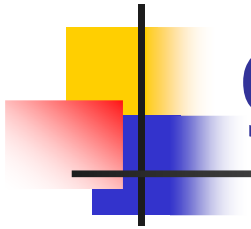
ofrece funciones de adaptación del estilo de interfaz



componentes de texto editables y no editables

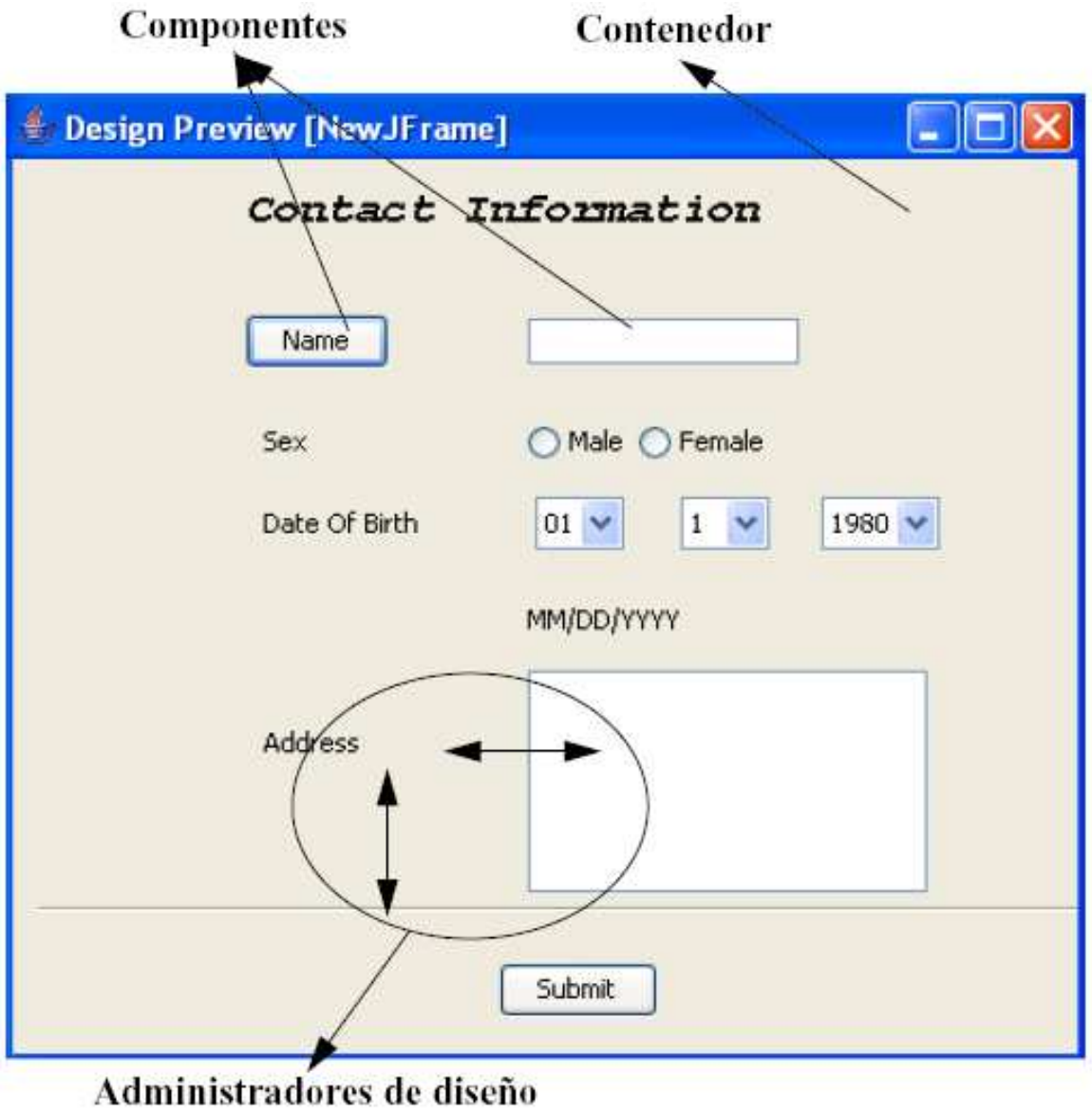
Componentes Visuales Básicos

Composición de una interfaz gráfica

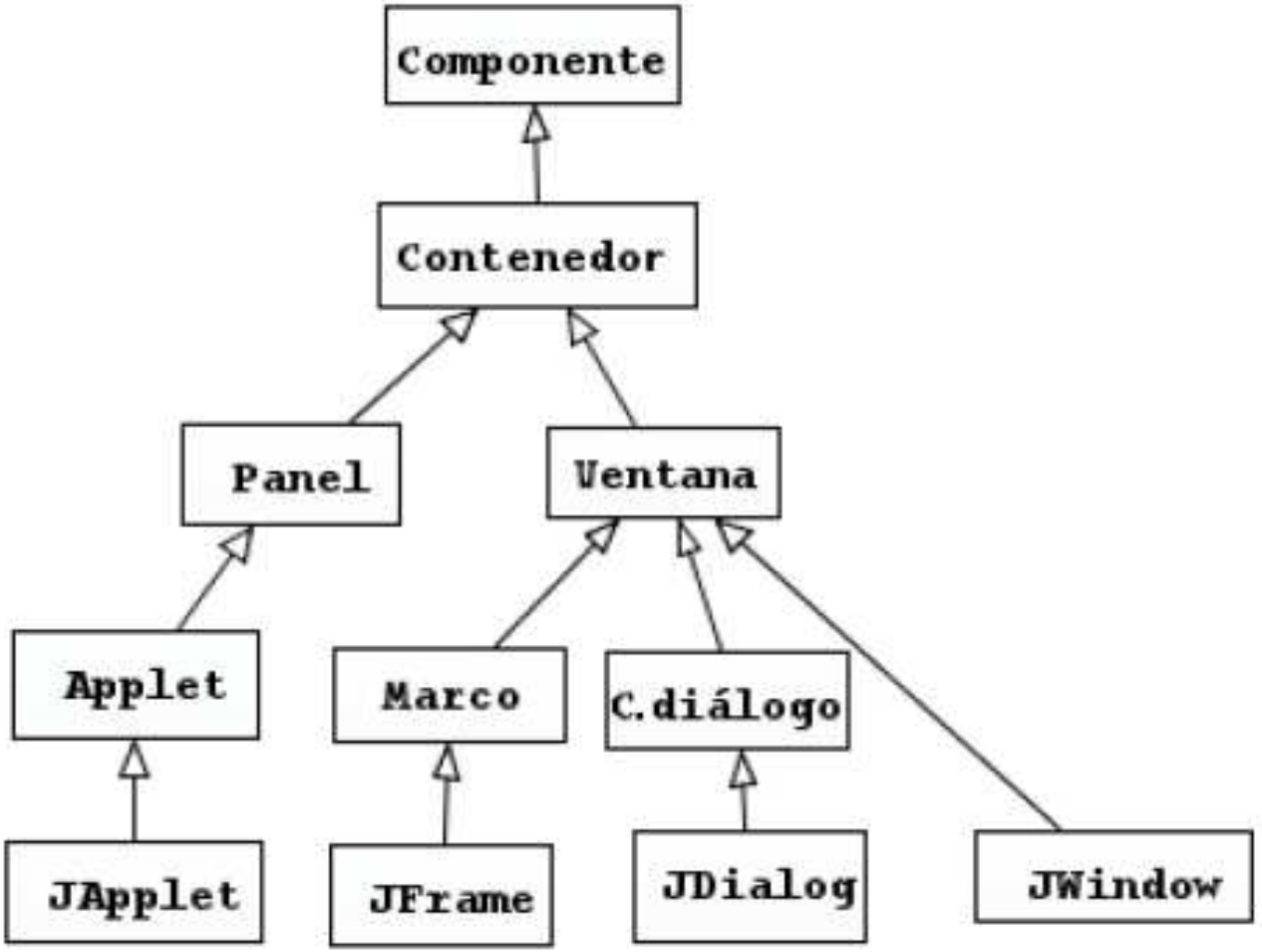


- Tres tipos de elementos:
  - Contenedores
    - Todos los componentes de una GUI se agregan a estos contenedores
    - JFrame, JDialog, JWindow y JApplet
  - Componentes
    - Derivan de la clase JComponent: JComboBox, JAbstractButton, JTextComponent, etc…
  - Administradores de diseño
    - BorderLayout, FlowLayout y GridLayout, etc…

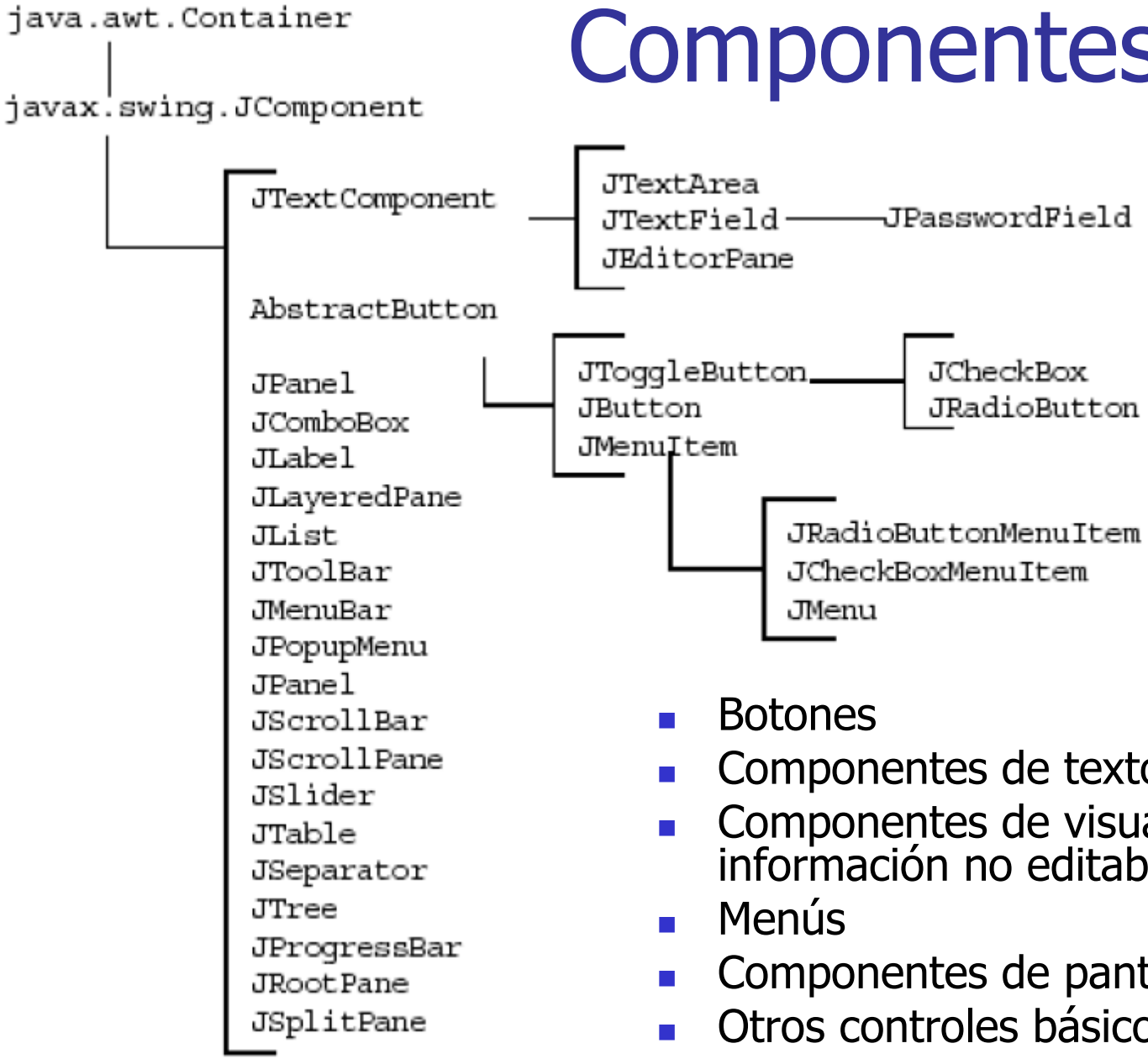
Elementos de la GUI



Contenedores Swing



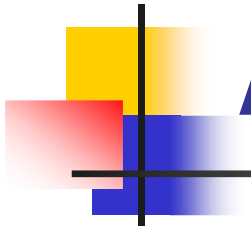
Componentes Swing



- Botones
- Componentes de texto
- Componentes de visualización de información no editable
- Menús
- Componentes de pantalla con formato
- Otros controles básicos

Propiedades de los componentes Swing

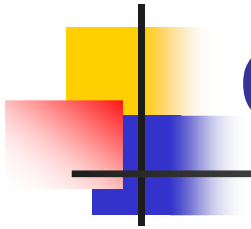
Propiedad	Métodos
Borde	`Border getBorder()` `void setBorder(Border b)`
Color de fondo y de primer plano	`void setBackground(Color bg)` `void setForeground(Color bg)`
Tipo de letra	`void setFont(Font f)`
Opacidad	`void setOpaque(boolean isOpaque)`
Tamaño máximo y mínimo	`void setMaximumSize(Dimension d)` `void setMinimumSize(Dimension d)`
Alineación	`void setAlignmentX(float ax)` `void setAlignmentY(float ay)`
Tamaño preferido	`void setPreferredSize(Dimension ps)`



Administradores de diseño

- Determinan el tamaño y la posición de los componentes dentro de un contenedor
- Alternativas:
  - Fijar pociones absolutas en pixels
    - Problemas de portabilidad entre plataformas y disposiitvos de visualización
  - Permitir redimensionamiento automático del interfaz
    - Realizado por el administrador de diseño

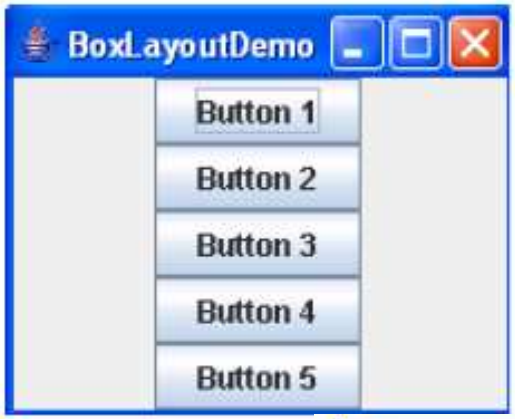
Misión de los administradores de diseño



- Gestionar el redimensionamiento de objetos de la interfaz gráfica por parte del usuario
- Permitir el uso de distintos tipos y tamaños de letra que hacen diferentes sistemas operativos o su personalización por parte del usuario
- Tratar los requisitos de disposición del texto que tienen las diferentes configuraciones internacionales (de izquierda a derecha, de derecha a izquierda, vertical)
- Cada administrador de diseño organiza los componentes según unas normas predeterminadas

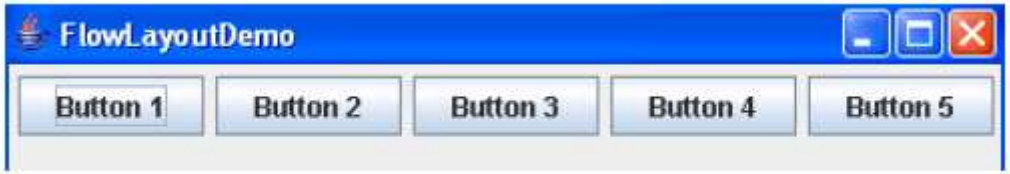
Administradores de diseño

Ejemplo de BorderLayout



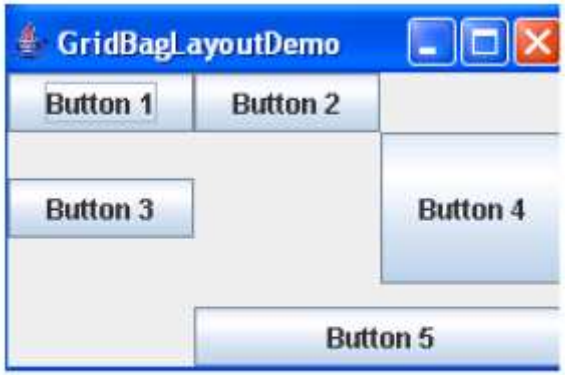
BoxLayout

GridLayout



Ejemplo de FlowLayout

Ejemplo de CardLayout



GridBagLayout

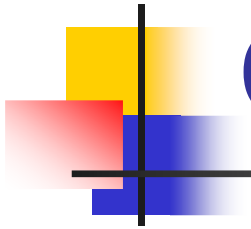
Adm. de diseño. Ejemplos

```
public class EjemploZonas {
    private JFrame f;
    private JButton bn, bs, bw, be, bc;

    public EjemploZonas() {
        f = new JFrame("Border Layout");
        bn = new JButton("Button 1");
        bc = new JButton("Button 2");
        bw = new JButton("Button 3");
        bs = new JButton("Button 4");
        be = new JButton("Button 5");
    }

    public void launchFrame() {
        f.add(bn, BorderLayout.NORTH);
        f.add(bs, BorderLayout.SOUTH);
        f.add(bw, BorderLayout.WEST);
        f.add(be, BorderLayout.EAST);
        f.add(bc, BorderLayout.CENTER);
        f.setSize(400,200);
        f.setVisible(true);
    }

    public static void main(String args[]) {
        EjemploZonas guiWindow2 = new EjemploZonas();
        guiWindow2.launchFrame();
    }
}
```

```
public void launchFrame() {
    f.setLayout(new FlowLayout());
    f.add(b1);
    f.add(b2);
    f.add(b3);
    f.add(b4);
    f.add(b5);
    f.pack();
    f.setVisible(true);
}
```

```
public void launchFrame() {
    f.setLayout (new GridLayout(3,2));
    f.add(b1);
    f.add(b2);
    f.add(b3);
    f.add(b4);
    f.add(b5);
    f.pack();
    f.setVisible(true);
}
```

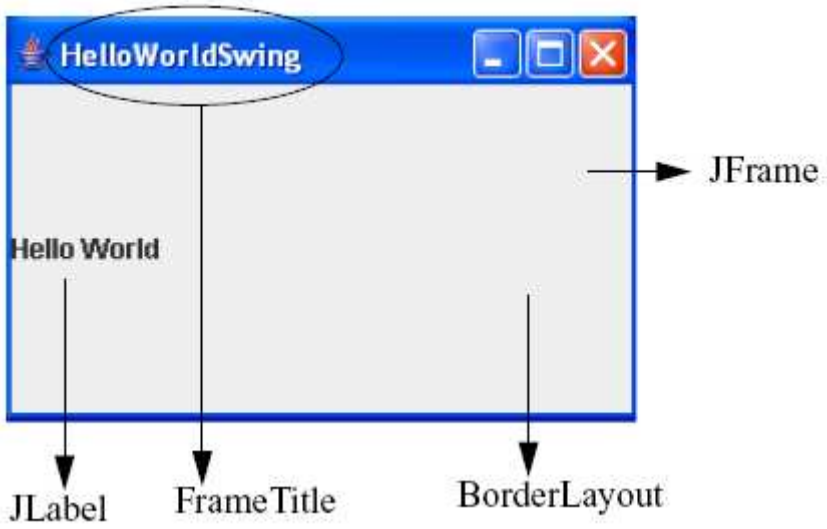
vi.e



Construcción de la GUI

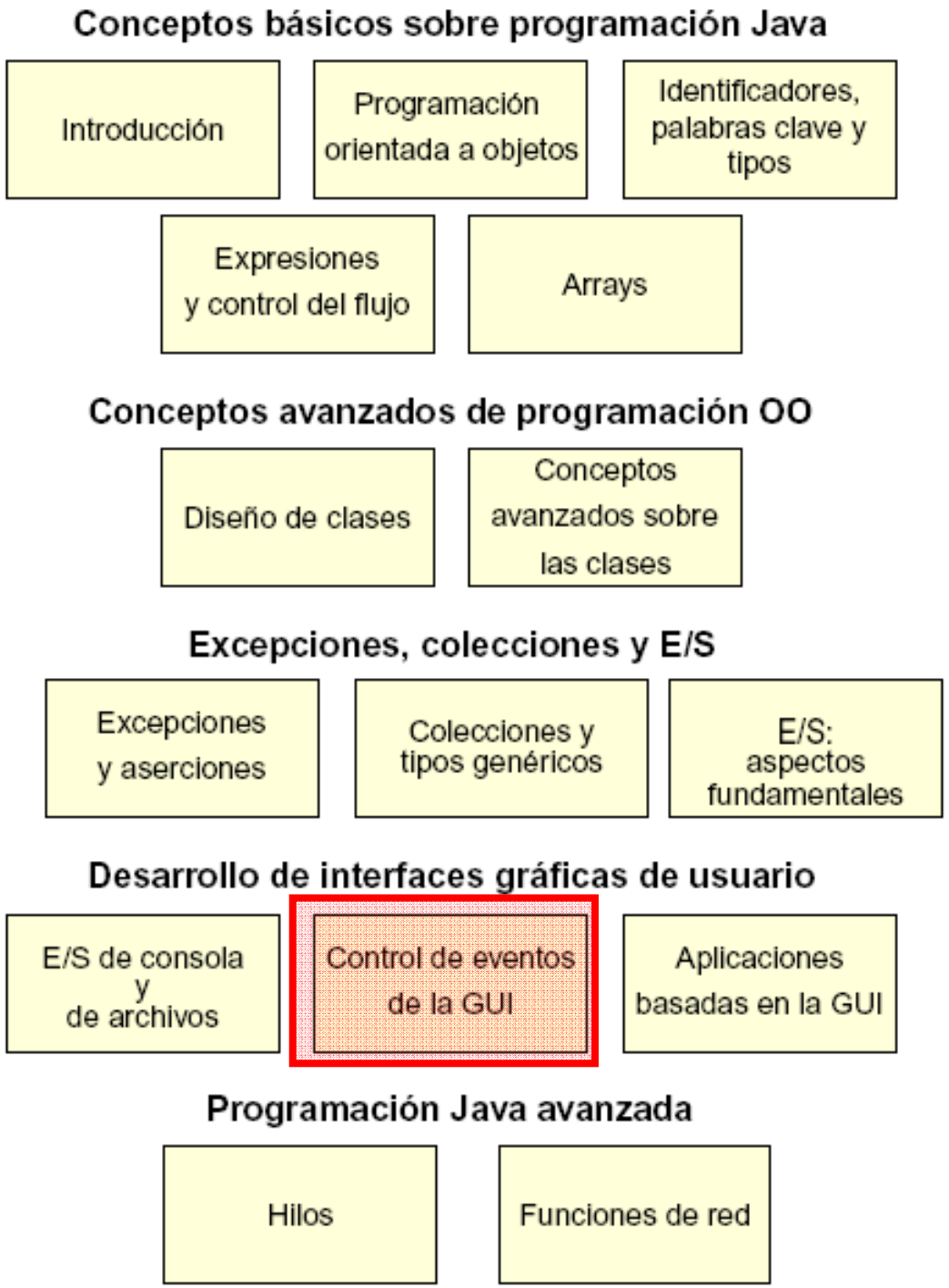
- Con herramientas de generación de código de interfaces
  - Borland, EclipseVE, MyEclipse, Matisse ...
  - (no se incluye en este curso)
- De forma programática
  - Útil en aprendizaje y experimentación
  - Programación poco eficiente, el código es muy tedioso y repetitivo

Aplicación HelloWorldSwing



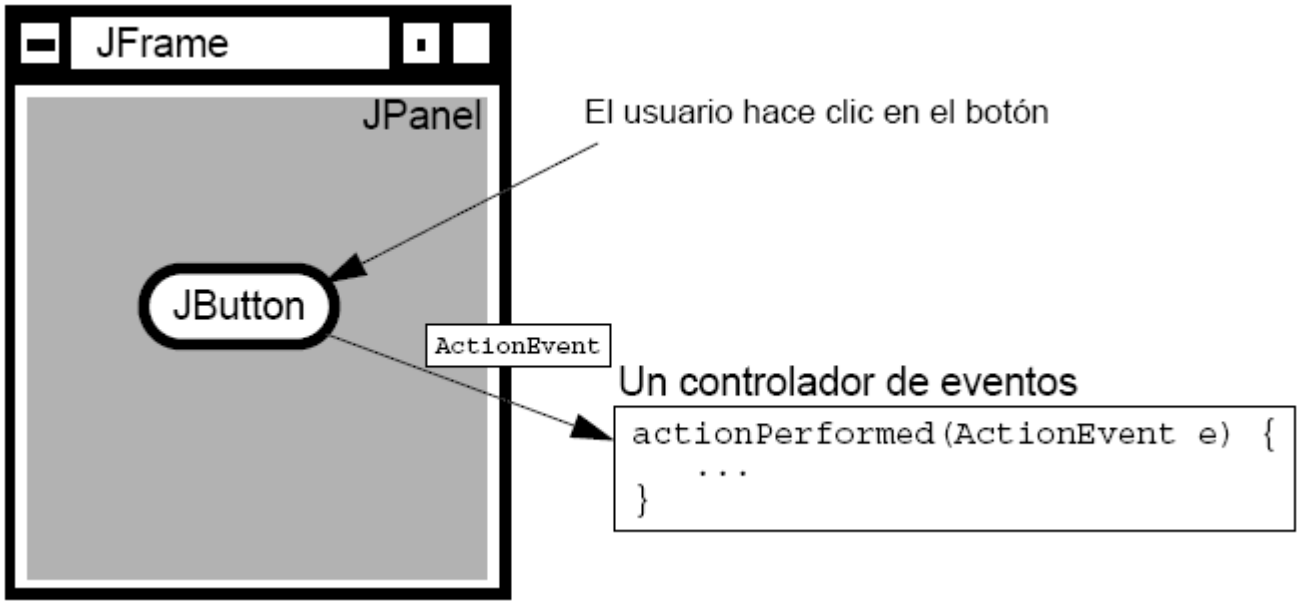

```
import javax.swing.*;
public class HelloWorldSwing {
    private static void createAndShowGUI() {
        JFrame frame = new JFrame("HelloWorldSwing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel("Hello World");
        frame.add(label);

        frame.setSize(300,200);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {createAndShowGUI();}
        });
    }
}
```

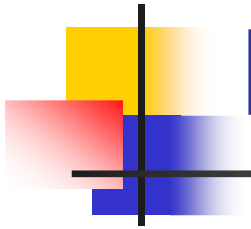
Contenidos


Conceptos básicos sobre programación Java
Introducción
Programación orientada a objetos
Identificadores, palabras clave y tipos
Expresiones y control del flujo
Arrays
Conceptos avanzados de programación OO
Diseño de clases
Conceptos avanzados sobre las clases
Excepciones, colecciones y E/S
Excepciones y aserciones
Colecciones y tipos genéricos
E/S: aspectos fundamentales
Desarrollo de interfaces gráficas de usuario
E/S de consola y de archivos
Control de eventos de la GUI
Aplicaciones basadas en la GUI
Programación Java avanzada
Hilos
Funciones de red

Qué es un evento

- Cualquier acción del usuario en la interfaz produce un evento
- El evento captura información acerca de lo ocurrido, en qué componente, cuando



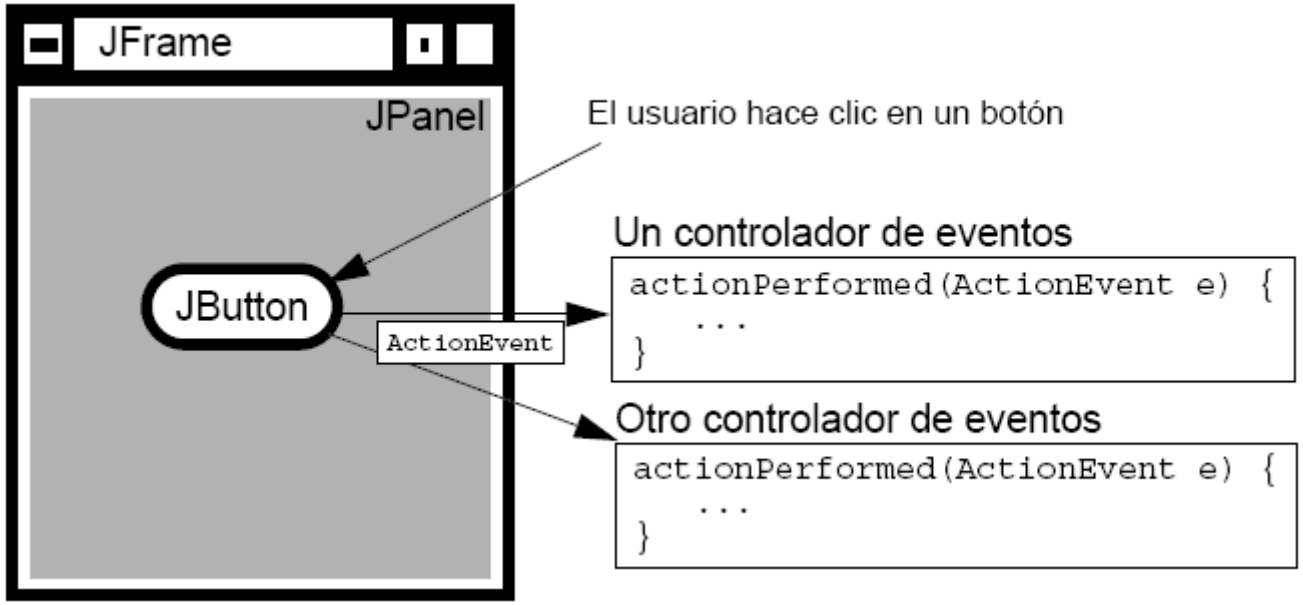


Manejo de los eventos

- Los eventos que afectan a la aplicación deben ser procesados
- Son procesados por un manejador de eventos (un método) que debe estar asociado a la fuente de eventos
- Un evento es un objeto que porta la información de la ocurrencia
  - getActionCommand
  - getModifiers
  - getWhen
  - paramString

Modelo de delegación

- Cada fuente de eventos, para cada evento, puede tener asignados varios manejadores de eventos
  - También se les llama *event listeners*



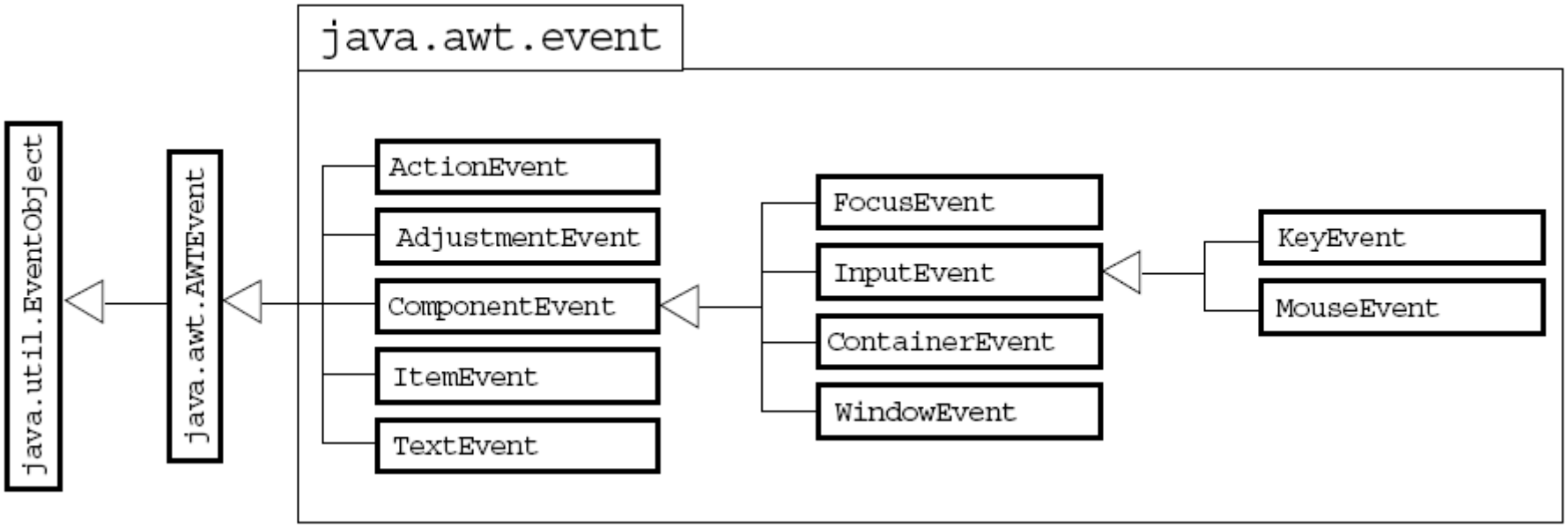
Ejemplo de receptor de eventos

Se podrían asociar varios listeners para el mismo evento

```
public class TestBoton {
    private JFrame f;
    private JButton b;

    public TestBoton() {
        f = new JFrame("Test");
        b = new JButton("Pulsar");
        b.setActionCommand("ButtonPressed");
    }

    public void launchFrame() {
        b.addActionListener(new ButtonHandler());
        f.add(b,BorderLayout.CENTER);
        f.pack();
        f.setVisible(true);
    }

    public static void main(String args[]) {
        TestBoton guiApp = new TestBoton();
        guiApp.launchFrame();
    }
}
```

```
public class ButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Se ha realizado una acción");
        System.out.println("El comando del botón es: " + e.getActionCommand());
    }
}
```

Categorías de eventos



- Para cada categoría de evento el receptor debe implementar una interfaz
  - La interfaz obliga a implementar uno o más métodos

Categorías de eventos, métodos e interfaces

Categoría	Nombre de la interfaz	Métodos
Acción	`ActionListener`	`actionPerformed(ActionEvent)`
Elemento	`ItemListener`	`itemStateChanged(ItemEvent)`

Ratón	`MouseListener`	`mousePressed(MouseEvent)`
		`mouseReleased(MouseEvent)`
		`mouseEntered(MouseEvent)`
		`mouseExited(MouseEvent)`
		`mouseClicked(MouseEvent)`

Movimiento del ratón	`MouseMotionListener`	`mouseDragged(MouseEvent)` `mouseMoved(MouseEvent)`

Eventos, métodos e interfaces...

Categoría	Nombre de la interfaz	Métodos

Tecla	`KeyListener`	`keyPressed(KeyEvent)`
		`keyReleased(KeyEvent)`
		`keyTyped(KeyEvent)`

Activación	`FocusListener`	`focusGained(FocusEvent)` `focusLost(FocusEvent)`
Ajuste	`AdjustmentListener`	`adjustmentValueChanged(AdjustmentEvent)`

Componente	`ComponentListener`	`componentMoved(ComponentEvent)`
		`componentHidden(ComponentEvent)`
		`componentResized(ComponentEvent)`
		`componentShown(ComponentEvent)`

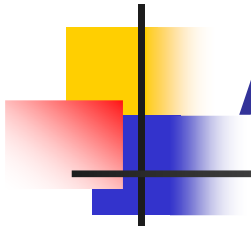
Ventana	`WindowListener`	`windowClosing(WindowEvent)`
		`windowOpened(WindowEvent)`
		`windowIconified(WindowEvent)`
		`windowDeiconified(WindowEvent)`
		`windowClosed(WindowEvent)`
		`windowActivated(WindowEvent)`
		`windowDeactivated(WindowEvent)`

Eventos, métodos e interfaces

Categoría	Nombre de la interfaz	Métodos
Contenedor	`ContainerListener`	`componentAdded(ContainerEvent)` `componentRemoved(ContainerEvent)`
Estado de ventana	`WindowStateListener`	`windowStateChanged(WindowEvent e)`
Activación de ventana	`WindowFocusListener`	`windowGainedFocus(WindowEvent e)` `windowLostFocus(WindowEvent e)`
Rueda del ratón	`MouseWheelListener`	`mouseWheelMoved(MouseWheelEvent e)`
Métodos de entrada	`InputMethodListener`	`caretPositionChanged (InputMethodEvent e)` `inputMethodTextChnaged (InputMethodEvent e)`
Jerarquía	`HierarchyListener`	`hierarchyChanged(HierarchyEvent e)`
Límites de la jerarquía	`HierarchyBoundsListener`	`ancestorMoved(HierarchyEvent e)` `ancestorResized(HierarchyEvent e)`
AWT	`AWTEventListener`	`eventDispatched(AWTEvent e)`
Texto	`TextListener`	`textValueChanged(TextEvent)`

Ejemplo complejo

```
public class TwoListener implements MouseMotionListener, MouseListener {
    private JFrame f;
    private JTextField tf;

    public TwoListener() {
        f = new JFrame("Ejemplo de dos receptor
        tf = new JTextField(30);
    }

    public void launchFrame() {
        JLabel label = new JLabel("Drag&drop");

        f.add(label, BorderLayout.NORTH);
        f.add(tf, BorderLayout.SOUTH);

        f.addMouseMotionListener(this);
        f.addMouseListener(this);

        f.setSize(300, 200);
        f.setVisible(true);
    }

    public void mouseDragged(MouseEvent e) {
        String s = "Arrastre del ratón: X = " + e.getX()
        + " Y = " + e.getY();
        tf.setText(s);
    }

    public void mouseEntered(MouseEvent e) {
        tf.setText("El ratón ha entrado");
    }

    public void mouseExited(MouseEvent e) {
        tf.setText("El ratón ha salido");
    }

    public void mouseMoved(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }
    public void mouseClicked(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }

    public static void main(String args[]) {
        TwoListener two = new TwoListener();
        two.launchFrame();
    }
}
```



Adaptadores de eventos

- En vez de implementar directamente las interfaces de los listener…
  - Lo que obliga a implementar todos los métodos aunque no se esté interesado
- … se pueden usar clases adaptadoras ya implementadas
  - Dan implementación por defecto (vacía) a todos los métodos

Adaptadores: ejemplo de uso

```
1   import java.awt.*;
2   import java.awt.event.*;
3   import javax.swing.*;
4
5   public class MouseClickHandler extends MouseAdapter {
6
7     // Sólo necesitamos el método mouseClicked, así que usamos
8     // un adaptador para no tener que escribir
9     // todos los métodos de control de eventos
10
11    public void mouseClicked(MouseEvent e) {
12      // Realiza acciones al pulsar el botón del ratón...
13    }
14  }
```

Control de eventos con clases internas

```
public class TestInner {
    private JFrame f;
    private JTextField tf;

    public TestInner() {
        f = new JFrame("Ejemplo de clases internas");
        tf = new JTextField(30);
    }

    class MyMouseMotionListener extends MouseMotionAdapter {
        public void mouseDragged(MouseEvent e) {
            String s = "Arrastre del ratón: X = "+ e.getX()
                       + " Y = " + e.getY();
            tf.setText(s);
        }
    }

    public void launchFrame() {
        JLabel label = new JLabel("Hacer clic y arrastrar el ratón");
        f.add(label, BorderLayout.NORTH);
        f.add(tf, BorderLayout.SOUTH);

        f.addMouseMotionListener(new MyMouseMotionListener());
        f.addMouseListener(new MouseClickHandler());

        f.setSize(300, 200);
        f.setVisible(true);
    }

    public static void main(String args[]) {
        TestInner obj = new TestInner();
        obj.launchFrame();
    }
}
```

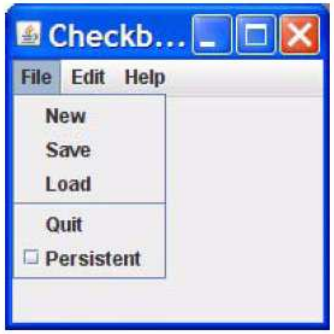
Control de eventos con clases anónimas

```
public class TestAnonymous {
    private JFrame f;
    private JTextField tf;

    public TestAnonymous() {
        f = new JFrame("Ejemplo de clases anónimas");
        tf = new JTextField(30);
    }

    public void launchFrame() {
        JLabel label = new JLabel("Hacer clic y arrastrar el ratón");
        f.add(label, BorderLayout.NORTH);
        f.add(tf, BorderLayout.SOUTH);

        f.addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent e) {
                String s = "Arrastre del ratón: X = "+ e.getX()
                           + " Y = " + e.getY();
                tf.setText(s);
            }
        }); // <-- paréntesis de cierre

        f.setSize(300, 200);
        f.setVisible(true);
    }

    public static void main(String args[]) {
    TestAnonymous obj = new TestAnonymous();
    obj.launchFrame();
    }
}
```

Creación de un menú

```
public TestMenu() {
    f = new JFrame("JMenuBar");
    mb = new JMenuBar();
    f.setJMenuBar(mb);

    m1 = new JMenu("File");
    m2 = new JMenu("Edit");
    m3 = new JMenu("Help");

    mi1 = new JMenuItem("New");
    mi2 = new JMenuItem("Save");
    mi3 = new JMenuItem("Load");
    mi4 = new JMenuItem("Quit");
    mi5 = new JCheckBoxMenuItem("Persistent");

    menuActionListener = new MenuActionListener();

    mi1.addActionListener(menuActionListener);
    mi2.addActionListener(menuActionListener);
    mi3.addActionListener(menuActionListener);
    mi4.addActionListener(menuActionListener);
    mi5.addActionListener(menuActionListener);

    mi4.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent arg0) {
            System.exit(0);
        }
    });

    mb.add(m1);
    mb.add(m2);
    mb.add(m3);

    m1.add(mi1);
    m1.add(mi2);
    m1.add(mi3);
    m1.addSeparator();
    m1.add(mi4);
    m1.add(mi5);
}
```

```
class MenuActionListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, e.getActionCommand());
    }
}
```



Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

Desarrollo de interfaces gráficas de usuario

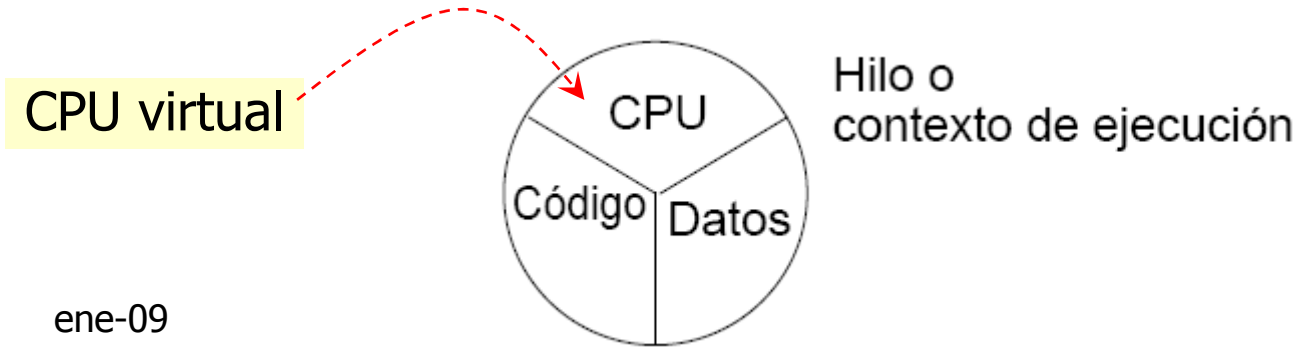
- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

Programación Java avanzada

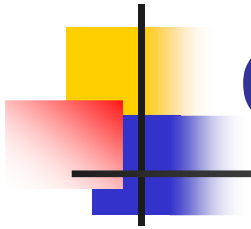
- Hilos
- Funciones de red

Hilos y clase Thread

- Hilo (o contexto de ejecución):
  - Encapsulamiento de una *CPU virtual* con su propio código y datos de programa
- java.lang.Thread:
  - Clase de Java que permite controlar los hilos (representa una CPU)

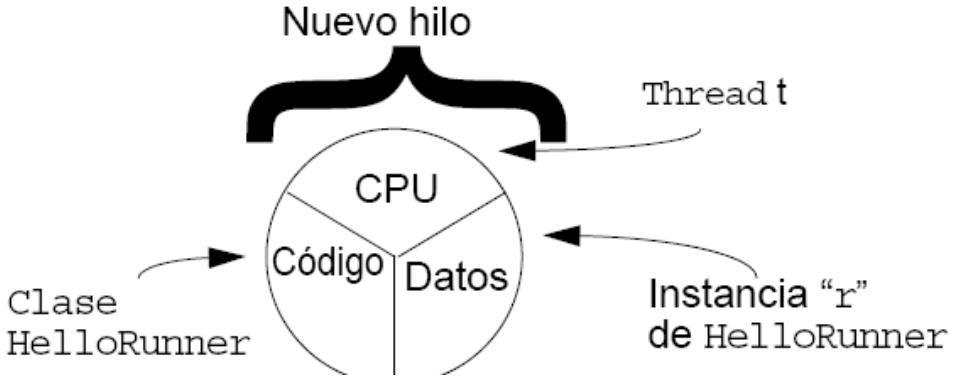


Hilos: CPU virtual, código y datos



- Varios hilos pueden compartir:
  - Código pero no datos
    - Cuando ejecutan código de distintas instancias de una clase
  - Datos pero no código
    - Cuando comparten el acceso a un mismo objeto
    - Requiere sincronización entre los hilos
- La CPU virtual se encapsula como una instancia de la clase Thread
  - En el constructor se le pasa el contexto
  - Un objeto: su código y sus datos son el contexto

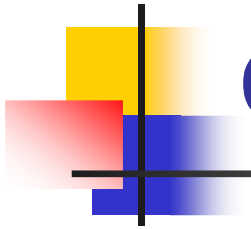
Creación del hilo




```
public class ThreadTester {
    public static void main(String args[]) {
        HelloRunner r = new HelloRunner();
        Thread t = new Thread(r);
        t.start();
    }
}

class HelloRunner implements Runnable {
    int i;

    public void run() {
        i = 0;
        while (true) {
            System.out.println("Hola " + i++);
            if (i == 50) {
                break;
            }
        }
    }
}
```

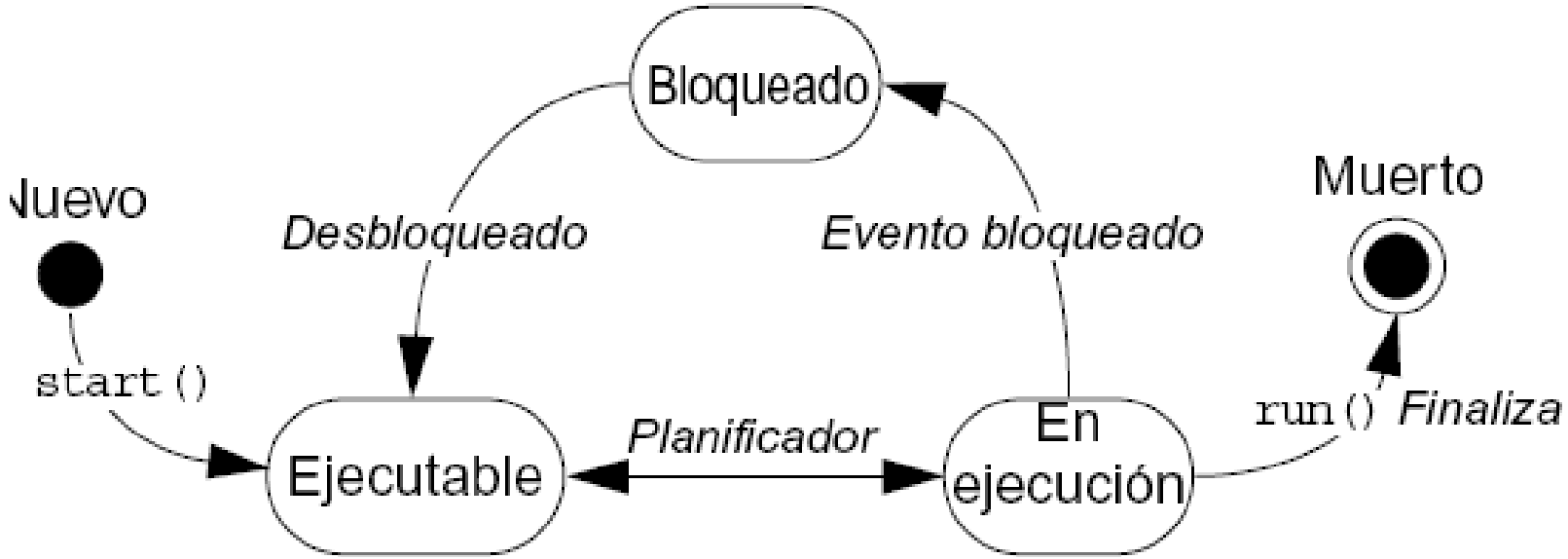
- Thread recibe una instancia de Runnable (public void run())

Planificación de la ejecución de hilos



- Preemptive pero no Round-Robin
  - Se ejecutan hasta que deja de ser ejecutable o hasta que otro hilo de mayor prioridad se pone en estado ejecutable.
- Deja de ser ejecutable cuando:
  - Thread.sleep()
  - Bloquea en espera de recurso
- Se agrupan en función de su prioridad

Estados de un hilo



Thread.sleep() permitir a otros hilos ejecutarse

```
1   public class Runner implements Runnable {
2     public void run() {
3       while (true) {
4         // hacer numerosas tareas
5         // ...
6         // Dar a otros hilos una oportunidad
7         try {
8           Thread.sleep(10);
9         } catch (InterruptedException e) {
10          // El periodo de espera de este hilo ha
11          // sido interrumpido por otro hilo
12        }
13      }
14    }
15  }
```

- Tiempo mínimo dormido (en milisegundos) a no ser que sea interrumpido su sueño

Fin de los hilos

```
public class Runner implements Runnable {
    private boolean timeToQuit=false;

    public void run() {
        while ( ! timeToQuit ) {
            // work until quit ...
        }
    }

    public void stopRunning() {
        timeToQuit=true;
    }
}
```

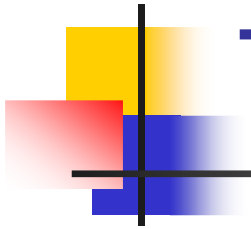
```
public class ThreadController {
    private Runner r = new Runner();
    private Thread t = new Thread(r);

    public void startThread() {
        t.start();
    }

    public void stopThread() {
        r.stopRunning();
    }
}
```

- Una vez terminado el hilo no se puede reanudar

Control básico de los hilos

- Estado del hilo
  - isAlive()
- Ajuste de la Prioridad
  - setPriority(int)
  - Un int entre MIN_PRIORITY y MAX_PRIORITY
- Espera por otro hilo
  - Thread.join()
- Permitir a otros ejecutarse
  - Thread.sleep(int milis)
  - Thread.yield()

```
Thread.MIN_PRIORITY
Thread.NORM_PRIORITY
Thread.MAX_PRIORITY
```



Thread.join() ejemplo

```
1   public static void main(String[] args) {
2     Thread t = new Thread(new Runner());
3     t.start();
4     ...
5     // Realizar tareas en paralelo con el otro hilo durante un tiempo
6     ...
7     // Esperar aquí hasta que termine el hilo en ejecución
8     try {
9       t.join();
10    } catch (InterruptedException e) {
11      // t ha vuelto antes de tiempo
12    }
13    ...
14    // Ahora continuar en este hilo
15    ...
16  }
```

Uso de synchronized

- Permite controlar hilos que comparten datos

```
public class MiPila {
    final int MAX = 10;
    int idx = 0;
    char [] data = new char[MAX];

    public void push(char c) {
        if (idx == MAX) return;
        data[idx] = c;
        idx++;
    }

    public char pop() {
        if (idx == 0) return 0;
        idx--;
        return data[idx];
    }
}
```

Problema con hilos compartiendo datos

```
public class Producer implements Runnable {
    private MiPila pila;

    public Producer (MiPila p) { pila = p; }

    public void run() {
        char c;

        for (int i = 0; i < ... {
            pila.push(c);
        }
    }
}
```

```
public class Consumer implements Runnable {
    private MiPila pila;

    public Consumer (MiPila p) {
        pila = p;
    }

    public void run() {
        char c;
        for (int i = 0; i < 200; i++) {
            c = pila.pop();
            // ...
        }
    }
}
```

Uso de la palabra clave synchronized

- En java cada objeto tiene un indicador de bloqueo asociado
- Synchronized adquiere ese bloqueo o espera a que esté libre
  - El bloqueo actúa como un semáforo
  - Synchonized maneja el semáforo

```
public class MiPila {
  ...
  public void push(char c) {
    synchronized(this) {
      data[idx] = c;
      idx++;
    }
  }
  ...
}
```

Se adquiere el bloqueo

Otros métodos no sinchronized sí tienen acceso a esos datos, habría que sincronizarlos todos

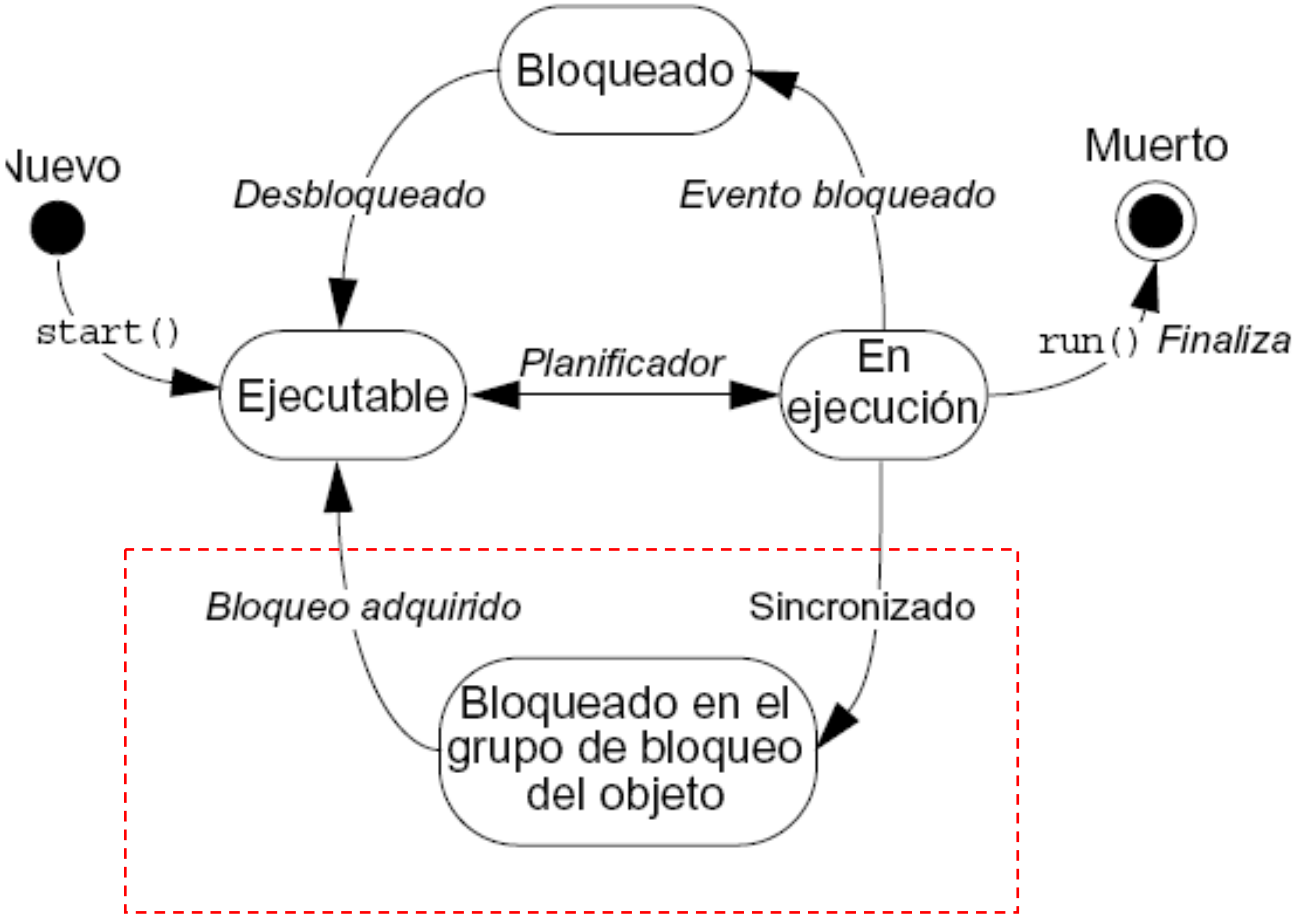
Synchronized, conclusiones

- El mecanismo de sincronización sólo funciona si *todo* el acceso a los datos "sensibles" es sincronizado
- Los datos de bloques sincronizados deberían ser private
- Métodos sincronized

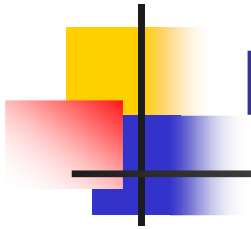
```
public void push(char c) {
  synchronized(this) {
    // Código del método push
  }
}
```

```
public synchronized void push(char c) {
  // Código del método push
}
```

Más estados para el hilo



Interacción de los hilos: wait y notify



- El hilo que llama a wait() queda a la espera de que otro hilo llame a notify() sobre el mismo objeto
  - Para hacer wait() o notify() se debe poseer el bloqueo → desde dentro de bloque synchronized
  - Al hacer wait() se libera el bloqueo
    - Lo cogerá uno de los hilos que hará notify()

Ejemplo de interacción de hilos

```
public class MiPila {
    final int MAX = 10;
    int idx = 0;
    char [] data = new char[MAX];

    public synchronized void push(char c) {
        if (idx == MAX) return;
        data[idx] = c;
        idx++;
        notify();
    }

    public synchronized char pop() {
        while (idx == 0){
            try {
                System.out.println("waiting");
                wait();
            } catch (InterruptedException e) { }
        }
        idx--;
        return data[idx];
    }
}
```

```
public class Main {

    public static void main(String[] args) {
        MiPila pila = new MiPila();
        Producer p1 = new Producer(pila);
        Producer p2 = new Producer(pila);
        Consumer c1 = new Consumer(pila);
        Consumer c2 = new Consumer(pila);

        Thread tp1 = new Thread(p1);
        Thread tp2 = new Thread(p2);
        Thread tc1 = new Thread(c1);
        Thread tc2 = new Thread(c2);

        tp1.start();
        tp2.start();
        tc1.start();
        tc2.start();
    }
}
```

Contenidos

Conceptos básicos sobre programación Java

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control del flujo
- Arrays

Conceptos avanzados de programación OO

- Diseño de clases
- Conceptos avanzados sobre las clases

Excepciones, colecciones y E/S

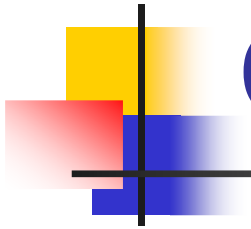
- Excepciones y aserciones
- Colecciones y tipos genéricos
- E/S: aspectos fundamentales

Desarrollo de interfaces gráficas de usuario

- E/S de consola y de archivos
- Control de eventos de la GUI
- Aplicaciones basadas en la GUI

Programación Java avanzada

- Hilos
- Funciones de red



Conexión en red: Sockets

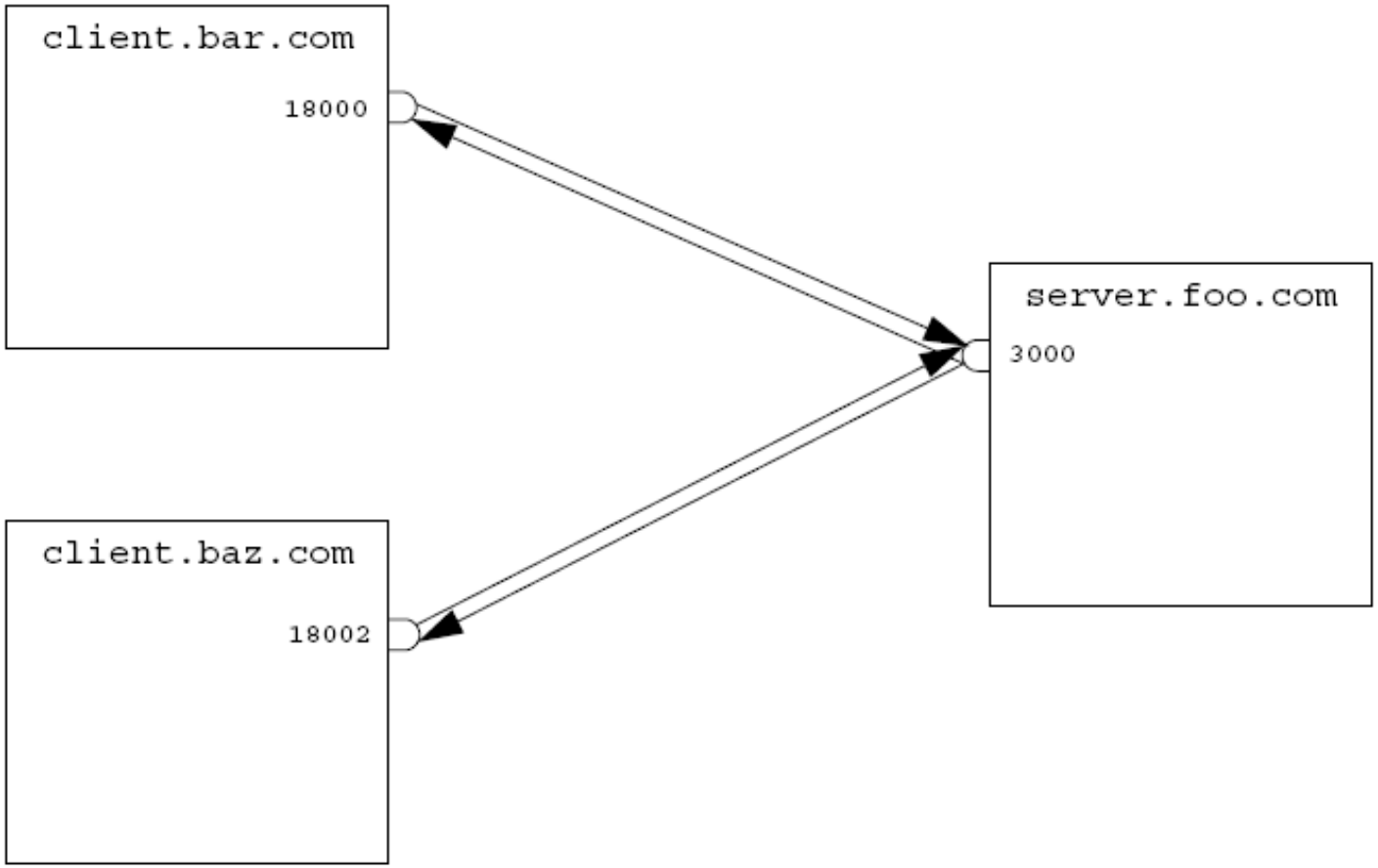
- Socket un modelo de programación de red
- Socket =

  IP cliente + Puerto Cliente +

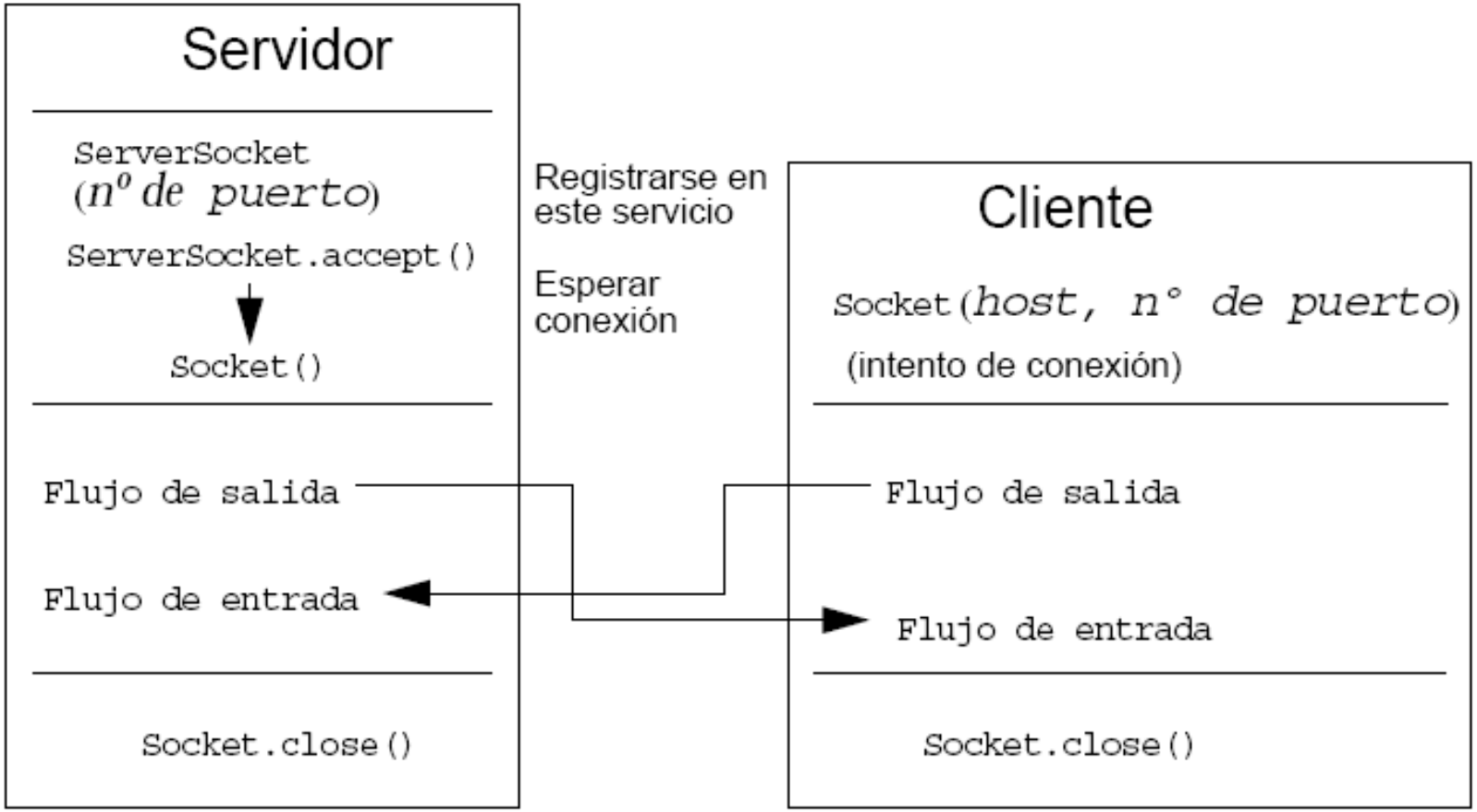
  IP servidor + Puerto servidor +

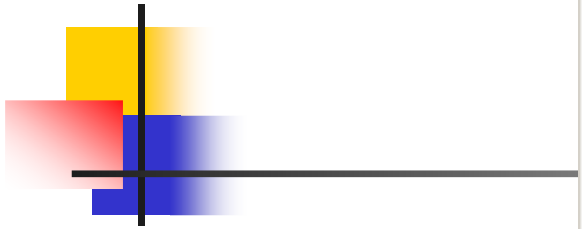
  protocolo

Esquema de conexión



Modelo de conexión en red de Java





Servidor TCP/IP mínimo

```
public class ServidorSimple {
    public static void main(String args[]) {
        ServerSocket s = null;

        try {
            s = new ServerSocket(5432);
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("Server started");

        while (true) {
            try {
                Socket s1 = s.accept();
                System.out.println("Connection acepted");

                BufferedWriter bw = new BufferedWriter(
                        new OutputStreamWriter(
                                s1.getOutputStream()));
                bw.write("Bienvenido a la red\n");
                bw.close();

                s1.close();
                System.out.println("Connection closed");
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Cliente TCP/IP mínimo

```
public class ClienteSimple {
    public static void main(String args[]) {
        try {
            Socket s1 = new Socket("127.0.0.1", 5432);

            BufferedReader br = new BufferedReader(
                    new InputStreamReader(s1.getInputStream()));
            System.out.println(br.readLine());
            br.close();

            s1.close();
        } catch (ConnectException connExc) {
            System.err.println("Imposible conectar.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```