Índice

1. Primer programa en Java

P1/HolaMundo.java

```
/* Este es un primer programa de prueba.
   Este archivo se llama "HolaMundo.java" */
class HolaMundo {
  // El programa comienza con una llamada a main().
  public static void main(String args[]) {
    System.out.println("Hola mundo.");
  }
}
```

1.1. Cuestiones sobre nomenclatura

- En Java, un fichero fuente contiene una o más definiciones de clase.
- El nombre de un fichero fuente suele ser el mismo de la clase que contenga.
- Los ficheros de programas en Java se guardarán con la extensión `.java`
- Debemos asegurarnos que coinciden mayúsculas y minúsculas en nombre de fichero y clase.

1.2. Compilación y ejecución del programa con CLDC

Los primeros programas que hacen uso sólo de características de CLDC los compilaremos y ejecutaremos usando el paquete `j2me_cldc-1_1` que está instalado en:

`$HOME/java/j2me_cldc`

Es conveniente crear un directorio temporal dónde se generarán las clases compiladas (ficheros .class).

```
mkdir tmpclasses
```

◊ **Compilación** Para compilar usaremos el compilador de J2SE (jdk1.5) que está instalado en las aulas de ordenadores en el directorio:

```
/usr/local/jdk1.5.0_06
```

```
javac -bootclasspath $CLDC_PATH/common/api/classes
      -d tmpclasses HolaMundo.java
```

- El compilador **javac** crea un archivo llamado `HolaMundo.class` en el directorio `tmpclasses` que contiene el bytecode compilado del programa.
- La opción `-bootclasspath` es usada para modificar el directorio del que se toman las clases básicas de Java en la compilación. Son utilizadas las librerías básicas de CLDC en lugar de las de J2SE.

◊ **Preverificación**

```
preverify -classpath $CLDC_PATH/common/api/classes:tmpclasses
          -d . HolaMundo
```

Esta orden crea un nuevo fichero `HolaMundo.class` a partir del fichero `HolaMundo.class` del directorio `tmpclasses`. En este caso es creado en el directorio actual.

◊ **Ejecución**

```
kvm -classpath . HolaMundo
```

◊ **Salida del programa**

```
Hola mundo
```

◊ Análisis del primer programa de prueba

- Comentarios de varias líneas

```
/*
  Este es un primer programa de prueba.
  Este archivo se llama "HolaMundo.java"
*/
```

- Definición de la clase

```
public class HolaMundo {
```

La definición de una clase, incluyendo todos sus miembros, estará entre la llave de apertura (`{`) y la de cierre (`}`).

- Comentario de una línea

```
// El programa comienza con una llamada a main().
```

- Cabecera del método main

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

  - Todas las aplicaciones Java comienzan su ejecución llamando a main. Sin embargo, cuando hacemos un Midlet, los programas no comienzan por main sino que ejecutan el constructor y luego el método `startApp()`.
  - La palabra **public** es un *especificador de acceso* (puede usarse fuera de la clase en la que se declara).
  - Otro especificador de acceso es el **private** (puede usarse sólo en métodos de la misma clase).
  - La palabra **static** permite que main sea llamado sin tener que crear un objeto de esta clase (HolaMundo).
  - La palabra **void** indica que main no devuelve ningún valor.
  - El parámetro **String args**[] declara una matriz de instancias de la clase **String** que corresponde a los argumentos de la línea de ordenes.

- Siguiente línea de código:

```
System.out.println("Hola mundo.");
```

Esta línea visualiza la cadena `"Hola mundo"` en la pantalla.

2. Tipos de datos, variables y matrices

- Java es un lenguaje fuertemente tipado:
  - Cada variable tiene un tipo, cada expresión tiene un tipo y cada tipo está definido estrictamente.
  - En todas las asignaciones (directas o a través de parámetros) se comprueba la compatibilidad de los tipos.
- Java es más estricto que C en asignaciones y paso de parámetros.

2.1. Tipos simples

- No son orientados a objetos y son análogos a los de C (por razones de eficiencia).
- Todos los tipos de datos tienen un **rango definido estrictamente** a diferencia de C (por razones de portabilidad).

◊ Enteros

Todos los tipos son enteros con signo.

- **byte**: 8 bits. $[-128, 127]$
- **short**: 16 bits. $[-32{,}768, 32{,}767]$
- **int**: 32 bits. $[-2{,}147{,}483{,}648, 2{,}147{,}483{,}647]$
- **long**: 64 bits. $[9{,}223{,}372{,}036{,}854{,}775{,}808, 9{,}223{,}372{,}036{,}854{,}775{,}807]$

◊ Tipos en coma flotante

Sólo disponibles en la versión 1.1 de CLDC.

- **float**: 32 bits. $[3,4 \cdot 10^{-38}, 3,4 \cdot 10^{38}]$
- **double**: 64 bits. $[1,7 \cdot 10^{-308}, 1,7 \cdot 10^{308}]$

◊ **Caracteres**

- Java utiliza código **Unicode** para representar caracteres.
- Es un conjunto de caracteres completamente internacional con todos los caracteres de todas las lenguas del mundo.
- **char**: 16 bits ($[0, 65536]$)
- Los caracteres ASCII van del $[0, 127]$ y el ISO-Latin-1 (caracteres extendidos) del $[0, 255]$

◊ **Booleanos**

boolean: Puede tomar los valores **true** o **false**

2.2. Literales

◊ **Enteros**

- Base decimal: 1, 2, 3, etc
- Base octal: 09
- Hexadecimal: Se antepone $0x$ o $0X$.
- Long: Se añade **L**. Ej: $101L$

◊ **Coma flotante**

Sólo disponibles en la versión 1.1 de CLDC. Son de doble precisión por defecto (double).

- Notación estándar: 2,0, 3,14, ,66
- Notación científica: $6{,}022E23$
- Añadiendo al final **F** se considera float, y añadiendo **D** double.

◊ **Booleanos**: **true** y **false**

◊ **Carácter**

- Se encierran entre comillas simples.
- Se pueden usar secuencias de escape:
  - `\141` (3 dígitos): Código en octal de la letra **a**
  - `\u0061` (4 dígitos): Código en hexadecimal (carácter Unicode) de la letra **a**
  - `\'`: Comilla simple
  - `\\`: Barra invertida
  - `\r`: Retorno de carro
  - `\t`: Tabulador `\b`: Retroceso

◊ **Cadena**: Entre comillas dobles

2.3. Variables

- La forma básica de declaración de variables es:

  ```
  tipo identificador [=valor][,identificador[=valor]...];
  ```

- `tipo` es un tipo básico, nombre de una clase o de un interfaz.
- Los inicializadores pueden ser dinámicos:

  ```
  double a=3.0, b=4.0;
  double c=Math.sqrt(a*a + b*b)
  ```

  El anterior código usa operaciones en coma flotante, que sólo están disponibles en CLDC 1.1
- Java permite declarar variables dentro de cualquier bloque.
- Java tiene dos **tipos de ámbito**
  - De clase:
  - De método:
- Los ámbitos se pueden anidar aunque las variables de un bloque interior no pueden tener el mismo nombre de alguna de un bloque exterior.
- Dentro de un bloque, las variables pueden declararse en cualquier momento pero sólo pueden usarse después de declararse.

2.4. Conversión de tipos

Funciona de forma parecida a C++

◊ Conversión automática de Java

- La **conversión automática de tipos** se hace si:
  - Los dos tipos son compatibles. Ejemplo: se puede asignar **int** a un **long**
  - El tipo destino es más grande que el tipo origen
- Los tipos **char** y **boolean** no son compatibles con el resto.

◊ Conversión de tipos incompatibles

Cuando queramos asignar a un tipo pequeño, otro mayor haremos uso de una conversión explícita:

```
int a;
byte b;
// ...
b = (byte) a;
```

◊ Promoción de tipo automática en expresiones

Además de las asignaciones, también se pueden producir ciertas conversiones automáticas de tipo en las expresiones.

- **short** y **byte** promocionan a **int** en expresiones para hacer los cálculos.
- Si un operador es **long** todo se promociona a **long**
- Si un operador es **float** todo se promociona a **float**
- Si un operador es **double** todo se promociona a **double**

Ejemplo de código con error de compilación:

```
byte b=50;
b = b*2; //Error, no se puede asignar un int a un byte
```

2.5. Vectores y matrices

Hay algunas diferencias en el funcionamiento de los vectores y matrices respecto a C y C++

2.5.1. Vectores

- **Declaración**

  `tipo nombre-vector[];`

  Esto sólo declara `nombre-vector` como vector de `tipo`, y le asigna **null**.

- **Reserva de la memoria**

  `nombre-vector=new tipo[tamaño]`

  Esto hace que se inicialicen a 0 todos los elementos.

- **Declaración y reserva al mismo tiempo**

  `int vector_int[]=new int[12];`

- **Inicialización al declarar el vector** `int vector_int[]={3,2,7}`

- El intérprete de Java comprueba siempre que no nos salimos de los índices del vector.

2.5.2. Matrices multidimensionales

- En Java las matrices son consideradas matrices de matrices.
- Declaración y reserva de memoria: `int twoD[][]=new int[4][5];`
- Podemos reservar cada fila de forma independiente:

```
int twoD[][]=new int[4][];
    twoD[0]=new int[1];
    twoD[1]=new int[2];
    twoD[2]=new int[3];
    twoD[3]=new int[2];
```

2.5.3. Sintaxis alternativa para la declaración de matrices

```
tipo[] nombre-matriz;
```

Ejemplo:

```
int[] a2=new int[3];
char[][] twoD2=new char[3][4];
```

2.6. Punteros

- Java no permite punteros que puedan ser accedidos y/o modificados por el programador, ya que eso permitiría que los applets rompieran el cortafuegos existente entre el entorno de ejecución y el host cliente.

3. Operadores

La mayoría de los operadores de Java funcionan igual que los de C/C++, salvo algunas excepciones (los operadores a nivel de bits de desplazamiento a la derecha).

- Los tipos enteros se representan mediante codificación en *complemento a dos*: los números negativos se representan invirtiendo sus bits y sumando 1.
- Desplazamiento a la derecha: `valor >> num`
  Ejemplo 1

  ```
  int a=35;
  a = a >> 2; // ahora a contiene el valor 8
  ```

  Si vemos las operaciones en binario:

  ```
  00100011      35
  >>2
  00001000      8
  ```

Este operador `>>` rellena el nuevo bit superior con el contenido de su valor previo:

Ejemplo 2

```
11111000      -8
>>2
11111100      -4
```

- Desplazamiento a la derecha sin signo: `valor >>> num`
  Rellena el bit superior con cero.

  Ejemplo

  ```
  int a=-1;
  a = a >> 24;
  ```

  Si vemos las operaciones en binario:

  ```
  11111111 11111111 11111111 11111111   -1 en binario como entero
  >>24
  00000000 00000000 00000000 11111111  255 en binario como entero
  ```

3.1. Tabla de precedencia de operadores:

```
.     []    ()
++    --    !      ~
*     /     % (se puede aplicar a flotantes)
+     -
<<    >>    >>>
<     >     <=     >=
==    !=
&
^
|
&&
||
?:
=     op=   *=   /=    %=    +=    -=    etc.)
```

4. Sentencias de control

De nuevo son prácticamente idénticas a las de C/C++ salvo en las sentencias `break` y `continue`

4.1. Sentencias de selección

- **if/else**

```
if( Boolean ) {
    sentencias;
}
else {
    sentencias;
}
```

- **switch**

```
switch( expr1 ) {
  case expr2:
       sentencias;
       break;
  case expr3:
       sentencias;
       break;
  default:
       sentencias;
       break;
  }
```

4.2. Sentencias de iteración

- **Bucles for**

```
for( expr1 inicio; expr2 test; expr3 incremento ) {
  sentencias;
}
```

- **Bucles while**

```
while( Boolean ) {
    sentencias;
}
```

- **Bucles do/while**

```
do {
    sentencias;
}while( Boolean );
```

4.3. Tratamiento de excepciones

- **Excepciones: try-catch-throw-throws-finally**

```
try {
    sentencias;
} catch( Exception ) {
    sentencias;
}
```

4.4. Sentencias de salto

Las sentencias `break` y `continue` se recomienda no usarlas, pues rompen con la filosofía de la programación estructurada:

4.4.1. break

Tiene tres usos:

- Para terminar una secuencia de sentencias en un **switch**
- Para salir de un bucle
- Como una forma de goto: `break etiqueta`

4.4.2. continue

Tiene dos usos:

- Salir anticipadamente de una iteración de un bucle (sin procesar el resto del código de esa iteración).
- Igual que antes pero se especifica una etiqueta para describir el bucle que lo engloba al que se aplica.

Ejemplo 1

```
uno: for( ) {
  dos: for( ){
         continue;     // seguiría en el bucle interno
         continue uno; // seguiría en el bucle principal
         break uno;    // se saldría del bucle principal
       }
}
```

4.4.3. return

Tiene el mismo uso de C/C++

Ejemplo 2

```
int func()
{
  if( a == 0 )
    return 1;
  return 0; // es imprescindible
}
```

5. Clases

5.1. Fundamentos

- **Clase**: La clase es la definición *general* de una entidad sobre la que estamos interesados en realizar algún tipo de procesamiento informático.

  Ejemplo: personas, coches, libros, alumnos, productos.

  La clase es la base de la PDO en Java.

- **Objeto**: Elemento *real* asociado a una clase (*instancia de una clase*).

5.1.1. Forma general de una clase

En Java, a diferencia de C++, la declaración de la clase y la implementación de los métodos se almacena en el mismo lugar:

```
class nombre_de_clase {
  tipo variable_de_instancia1;
  // ...
  tipo variable_de_instanciaN;
  tipo nombre_de_método1(lista_de_parámetros) {
    // cuerpo del método
  }
  // ...
  tipo nombre_de_métodoN(lista_de_parámetros) {
    // cuerpo del método
  }
}
```

5.1.2. Una clase sencilla

```
class Box {
  double width;
  double height;
  double depth;
}
```

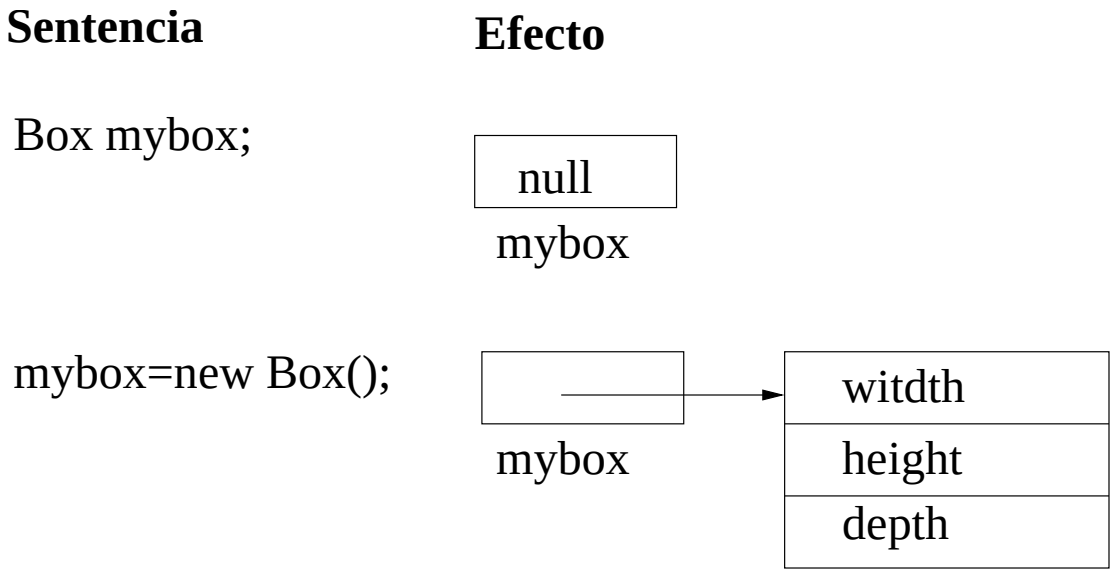
5.2. Declaración de objetos

- Para crear objetos de una clase se dan dos pasos:
  - Declarar una variable del tipo de la clase.
  - Obtener una copia física y real del objeto con el operador **new** asignándosela a la variable.

```
Box mybox=new Box();
```

o bien

```
Box mybox; // declara la referencia a un objeto
mybox=new Box(); // reserva espacio para el objeto
```



5.2.1. Operador new

- El operador **new** reserva memoria dinámicamente para un objeto.

  ```
  variable = new nombre_de_clase();
  ```

- También hace que se llame al constructor de la clase. En este caso se llama al *constructor por defecto*.

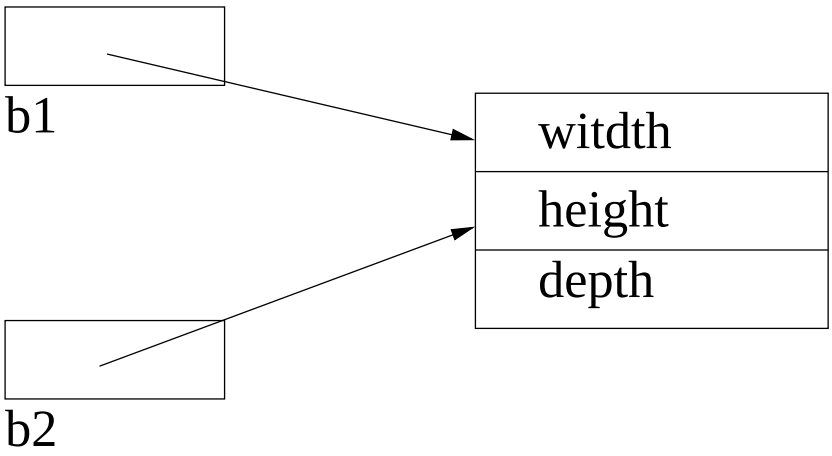
P2/BoxDemo.java

```
/* Un programa que utiliza la clase Box
   Este archivo se debe llamar BoxDemo.java */
class Box {
  double width;
  double height;
  double depth;
}
// Esta clase declara un objeto del tipo Box
class BoxDemo {
  public static void main(String args[]) {
    Box mybox = new Box();
    double vol;
    // asigna valores a las variables de instancia de mybox
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;
    // calcula el volumen de la caja
    vol = mybox.width * mybox.height * mybox.depth;
    System.out.println("El volumen es " + vol);
  }
}
```

5.3. Asignación de variables referencia a objeto

- Las variables referencia a objeto se comportan de manera diferente a lo que se podría esperar en asignaciones.

```
Box b1 = new Box();
Box b2 = b1;
```



5.4. Métodos

- Como dijimos anteriormente dentro de las clases podemos tener variables de instancia y métodos.
- La sintaxis para definir un método es la misma de C y C++:

```
tipo nombre_de_método(lista_de_parámetros){
  // cuerpo del método
}
```

- Dentro de los métodos podemos usar las variables de instancia directamente.
- Para llamar a un método usamos:

```
objeto.nombre_metodo(parametros);
```

P3/BoxDemo4.java

```
/* Un programa que utiliza la clase Box
   Este archivo se debe llamar BoxDemo4.java
*/
class Box {
  double width;
  double height;
  double depth;
  // calcula y devuelve el volumen
  double volume() {
    return width * height * depth;
  }
}

// Esta clase declara un objeto del tipo Box
class BoxDemo4 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;
    // asigna valores a las variables de instancia de mybox
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    vol = mybox1.volume();
    System.out.println("El volumen es " + vol);
    vol = mybox2.volume();
    System.out.println("El volumen es " + vol);
  }
}
```

5.4.1. Métodos con parámetros

P4/BoxDemo5.java

```
/* Un programa que utiliza la clase Box */
class Box {
  double width;
  double height;
  double depth;
  // calcula y devuelve el volumen
  double volume() {
    return width * height * depth;
  }
  // establece las dimensiones de la caja
  void setDim(double w, double h, double d) {
    width = w;
    height= h;
    depth = d;
  }
}
// Esta clase declara un objeto del tipo Box
class BoxDemo5 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // inicializa cada caja
    mybox1.setDim(10,20,15);
    mybox2.setDim(3,6,9);
    vol = mybox1.volume();
    System.out.println("El volumen es " + vol);
    vol = mybox2.volume();
    System.out.println("El volumen es " + vol);
  }
}
```

5.5. Constructores

- El constructor inicializa el objeto inmediatamente después de su creación.
- Tiene el mismo nombre que la clase, y no devuelve nada (ni siquiera `void`).
- Cuando no especificamos un constructor, Java crea un *constructor por defecto* , que inicializa todas las variables de instancia a cero.

P5/BoxDemo6.java

```
class Box {
  double width;
  double height;
  double depth;
  Box() {
    System.out.println("Constructor de Box");
    width = 10;
    height= 10;
    depth = 10;
  }
  double volume() {
    return width * height * depth;
  }
}
class BoxDemo6 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;
    vol = mybox1.volume();
    System.out.println("El volumen es " + vol);
    vol = mybox2.volume();
    System.out.println("El volumen es " + vol);
  }
}
```

5.5.1. Constructores con parámetros

P6/BoxDemo7.java

```
class Box {
  double width;
  double height;
  double depth;
  Box() {
    System.out.println("Constructor de Box");
    width = 10;
    height= 10;
    depth = 10;
  }
  Box(double w,double h,double d) {
    width = w;
    height = h;
    depth = d;
  }
  double volume() {
    return width * height * depth; }
}
class BoxDemo7 {
  public static void main(String args[]) {
    Box mybox1 = new Box(10,20,15);
    Box mybox2 = new Box(3,6,9);
    Box mybox3 = new Box();
    double vol;
    vol = mybox1.volume();
    System.out.println("El volumen es " + vol);
    vol = mybox2.volume();
    System.out.println("El volumen es " + vol);
    vol = mybox3.volume();
    System.out.println("El volumen es " + vol);
  }
}
```

5.6. this

- Los métodos pueden referenciar al objeto que lo invocó con la palabra clave **this**.

  Ejemplo de su utilidad

```
Box(double w,double h,double d) {
    this.width = width;
    this.height = height;
    this.depth = depth;
  }
```

5.7. Recogida de basura

- Java libera los objetos de manera automática cuando ya no hay ninguna referencia a esos objetos.
- En J2SE tenemos en la clase `Object` el método `void finalize()`, pero no en CLDC. Tal método es llamado automáticamente por el recolector de basura cuando no existe ninguna referencia al objeto.
- Cuando no necesitemos un objeto en CLDC es conveniente asignarle el valor `null` a la variable que lo apunta para que el recolector de basura pueda actuar.

5.8. Ejemplo de clase (Clase Stack): P7/TestStack.java

- La clase permite el encapsulamiento de datos y código.
- La clase es como una *caja negra*: No hace falta saber que ocurre dentro para poder utilizarla.

```
class Stack {
  int stck[] = new int[10];
  int tos;
  Stack() {/*Inicializa la posición superior de la pila*/
    tos = -1;
  }
  void push(int item) {/*Introduce un elemento en la pila*/
    if(tos==9)
      System.out.println("La pila esta llena");
    else
      stck[++tos] = item;
  }
  int pop() {/*Extrae un elemento de la pila*/
    if(tos < 0) {
      System.out.println("La pila esta vacía");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();
    // introduce algunos números en la pila
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);
    // extrae los números de la pila
    System.out.println("Contenido de la pila mystack1:");
    for(int i=0; i<10; i++)
       System.out.println(mystack1.pop());
    System.out.println("Contenido de la pila mystack2:");
    for(int i=0; i<10; i++)
       System.out.println(mystack2.pop());
  }
}
```

6. Métodos y clases

6.1. Sobrecarga de métodos

Consiste en que dos o más métodos de una clase tienen el mismo nombre, pero con listas de parámetros distintos.

- La sobrecarga es usada en Java para implementar el *polimorfismo*.
- Java utiliza el tipo y/o el número de argumentos como guía para ver a cual método llamar.
- El tipo que devuelve un método es insuficiente para distinguir dos versiones de un método.

P8/Overload.java

```
class OverloadDemo {
  void test() {
    System.out.println("Sin parametros");
  }

  // Sobrecarga el metodo test con un parámetro entero
 void test(int a) {
    System.out.println("a: " + a);
  }

  // Sobrecarga el metodo test con dos parametros enteros
  void test(int a, int b) {
    System.out.println("a y b: " + a + " " + b);
  }

  // Sobrecarga el metodo test con un parámetro double
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}
```

```
class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // llama a todas las versiones de test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.2);
    System.out.println("Resultado de ob.test(123.2): " + result);
  }
}
```

La salida del programa sería:

```
Sin parametros
a: 10
a y b: 10 20
a double: 123.2
Resultado de ob.test(123.2): 15178.2
```

6.1.1. Sobrecarga con conversión automática de tipo

Java busca una versión del método cuyos parámetros actuales coincidan con los parámetros formales, en número y tipo. Si el tipo no es exacto puede que se aplique *conversión automática de tipos*.

```
class OverloadDemo {
  void test() {
    System.out.println("Sin parametros");
  }

  // Sobrecarga el metodo test con dos parametros enteros
 void test(int a, int b) {
    System.out.println("a y b: " + a + " " + b);
  }

  // Sobrecarga el metodo test con un parámetro double
  void test(double a) {
    System.out.println("Dentro de test(double) a: " + a);
  }
}

class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;

    ob.test();
    ob.test(10, 20);

    ob.test(i); // esto llama a test(double)
    ob.test(123.2); // esto llama a test(double)
  }
}
```

6.1.2. Sobrecarga de constructores

Además de los métodos normales, también pueden sobrecargarse los constructores.

P9/OverloadCons.java

```
class Box {
  double width;
  double height;
  double depth;

  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
  Box() {
    width = -1;  // utiliza -1 para indicar
    height = -1; // que una caja no está
    depth = -1;  // inicializada
  }
  Box(double len) {
    width = height = depth = len;
  }

  double volume() {
    return width * height * depth;
  }
}
```

```
class OverloadCons {
  public static void main(String args[]) {
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    double vol;

    vol = mybox1.volume();
    System.out.println("El volumen de mybox1 es " + vol);
    vol = mybox2.volume();
    System.out.println("El volumen de mybox2 es " + vol);
    vol = mycube.volume();
    System.out.println("El volumen de mycube es " + vol);
  }
}
```

6.2. Objetos como parámetros

Un objeto de una determinada clase puede ser parámetro de un método de esa u otra clase.

P10/PassOb.java

```
class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // devuelve true si o es igual al objeto llamante
  boolean equals(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}

class PassOb {
  public static void main(String args[]) {
    Test ob1 = new Test(100, 22);
    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, -1);

    System.out.println("ob1 == ob2: " + ob1.equals(ob2));
    System.out.println("ob1 == ob3: " + ob1.equals(ob3));
  }
}
```

Uno de los ejemplos más usuales es en constructores para hacer copias de objetos:

```
  Box(Box ob) {
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

Box mybox = new Box(10,20,15);
Box myclone = new Box(mybox);
```

6.3. Paso de argumentos

- El paso de tipos simples a los métodos es siempre *por valor*.
- Los objetos se pasan siempre *por referencia*.

P11/CallByValue.java

```
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
class CallByValue {
  public static void main(String args[]) {
    Test ob = new Test();
    int a = 15, b = 20;
    System.out.println("a y b antes de la llamada:"+a+" "+b);
    ob.meth(a, b);
    System.out.println("a y b despues de la llamada:"+a+" "+b);
  }
}


class Test {
  int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
  // pasa un objecto
  void meth(Test o) {
    o.a *=  2;
    o.b /= 2;
  }
}
```

P12/CallByRef.java

```
class CallByRef {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);
    System.out.println("ob.a y ob.b antes de la llamada: " +
                       ob.a + " " + ob.b);
    ob.meth(ob);
    System.out.println("ob.a y ob.b después de la llamada: " +
                       ob.a + " " + ob.b);
  }
}
```

6.4. Control de acceso

Los especificadores de acceso para variables de instancia y métodos son:

- **private**: Sólo es accesible por miembros de la misma clase.
- **public**: Accesible por miembros de cualquier clase.
- **protected**: Está relacionado con la herencia.
- Por defecto: si no se indica nada los miembros son públicos dentro de su mismo paquete

P13/AccessTest.java

```
class Test {
  int a; // acceso por defecto
  public int b; // acceso publico
  private int c; // acceso privado
  void setc(int i) { // establece el valor de c
    c = i;
  }
  int getc() { // obtiene el valor de c
    return c;
  }
}
```

```
class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();

    // Esto es correcto, a y b pueden ser accedidas directamente
    ob.a = 10;
    ob.b = 20;

    // Esto no es correcto y generara un error de compilación
//  ob.c = 100; // Error!

    // Se debe acceder a c a través de sus métodos
    ob.setc(100); // OK

    System.out.println("a, b, y c: " + ob.a + " " +
                       ob.b + " " + ob.getc());
  }
}
```

6.5. Especificador static

- Un miembro **static** puede ser usado sin crear ningún objeto de su clase.
- Puede aplicarse a variables de instancia y métodos
  - Las variables **static** son variables globales. Existe una sola copia de la variable para todos los objetos de su clase.
  - Los métodos **static**
    - Sólo pueden llamar a métodos **static**
    - Sólo deben acceder a datos **static**
    - No pueden usar **this** o **super**
- Una clase puede tener un bloque static que se ejecuta una sola vez cuando la clase se carga por primera vez.

P14/UseStatic.java

```
class UseStatic {
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
  static {
    System.out.println("Bloque estático inicializado.");
    b = a * 4;
  }
  public static void main(String args[]) {
    meth(42);
  }
}
```

6.6. Especificador final con variables

Una variable **final** es una variable en la que no podemos modificar su contenido.

- Las variables final deben inicializarse al declararlas.
- Son similares al **const** de C/C++
- Suele utilizar identificadores en mayúscula:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

- Existe una sola copia de ellas para todos los objetos de la clase.
- Las variables final suelen ser también static.

6.7. Clase String

Es una clase muy usada, que sirve para almacenar cadenas de caracteres (incluso los literales).

- Los objetos **String** no pueden modificar su contenido.
- Los objetos **StringBuffer** si que pueden modificarse.
- El operador + permite concatenar cadenas.
- El método **boolean equals(String objeto)** compara si dos cadenas son iguales.
- El método **int length()** obtiene la longitud de una cadena.
- El método **char charAt(int pos)** obtiene el carácter que hay en la posición **pos**.

P15/StringDemo2.java

```
// Muestra la utilización de algunos metodo de la clase String
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "Primera cadena";
    String strOb2 = "Segunda cadena";
    String strOb3 = strOb1;
    System.out.println("La longitud de strOb1 es: " +
                       strOb1.length());
    System.out.println("El carácter de la posición 3 de strOb1 es: " +
                       strOb1.charAt(3));
    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");
    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```

6.8. Argumentos de la línea de ordenes

P16/CommandLine.java

```
// Presenta todos los argumentos de la linea de ordenes
class CommandLine {
  public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +
                         args[i]);
  }
}
```

Si ejecutamos este programa con:

```
kvm -classpath . CommandLine esto es una prueba 100 -1
```

obtendremos como salida:

```
args[0]: esto
args[1]: es
args[2]: una
args[3]: prueba
args[4]: 100
args[5]: -1
```

7. Herencia

La herencia es uno de los pilares de la PDO ya que permite la creación de clasificaciones jerárquicas.

- Permite definir una clase general que define características comunes a un conjunto de elementos relacionados.
- Cada subclase puede añadir aquellas cosas particulares a ella.
- Las subclases heredan todas las variables de instancia y los métodos definidos por la superclase, y luego pueden añadir sus propios elementos.

7.1. Fundamentos

- Para definir que una subclase hereda de otra clase usamos **extends**.
- Java no permite la herencia múltiple.
- La subclase no puede acceder a aquellos miembros declarados como **private** en la superclase.

P17/DemoBoxWeight.java

```
// Este programa utiliza la herencia para extender la clase Box.
class Box {
  double width;
  double height;
  double depth;
  Box(Box ob) {
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
```

```
  Box() {
    width = -1;
    height = -1;
    depth = -1;
  }
  Box(double len) {
    width = height = depth = len;
  }
  double volume() {
    return width * height * depth;
  }
}
class BoxWeight extends Box {
  double weight; // peso de la caja
  BoxWeight(double w, double h, double d, double m) {
    width = w;
    height = h;
    depth = d;
    weight = m;
  }
}
class DemoBoxWeight {
  public static void main(String args[]) {
    BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
    BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
    double vol;
    vol = mybox1.volume();
    System.out.println("Volumen de mybox1 es " + vol);
    System.out.println("Peso de mybox1 es " + mybox1.weight);
    System.out.println();
    vol = mybox2.volume();
    System.out.println("Volumen de mybox2 es " + vol);
    System.out.println("Peso de mybox2 es " + mybox2.weight);
  }
}
```

7.1.1. Una variable de la superclase puede referenciar a un objeto de la subclase

El tipo de la variable referencia, (y no el del objeto al que apunta) es el que determina los miembros que son accesibles.

P18/RefDemo.java

```
class RefDemo {
  public static void main(String args[]) {
    BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
    Box plainbox = new Box();
    double vol;

    vol = weightbox.volume();
    System.out.println("El volumen de weightbox ex " + vol);
    System.out.println("El peso de weightbox es " +
                       weightbox.weight);
    System.out.println();

    // asigna una referencia de BoxWeight a una referencia de Box
    plainbox = weightbox;
    vol = plainbox.volume(); // OK,  volume() definido en Box
    System.out.println("Volumen de plainbox es " + vol);

    /* La siguiente sentencia no es válida porque plainbox
       no define un miembro llamado weight. */
//  System.out.println("El peso de plainbox es " +
                       //  plainbox.weight);
  }
}
```

7.2. Uso de super

La palabra reservada **super** permite a una subclase referenciar a su superclase inmediata. Es utilizada en las siguientes situaciones:

1. Para llamar al constructor de la superclase desde el constructor de la subclase.

   En este caso `super()` debe ser la primera sentencia ejecutada dentro del constructor.

2. Para acceder a un miembro de la superclase que ha sido ocultado por un miembro de la subclase.

Ejemplo de uso en caso 1

```
class BoxWeight extends Box {
  double weight;

  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // llama al constructor de la superclase
    weight = m;
  }
  BoxWeight(BoxWeight ob) {
    super(ob);
    weight = ob.weight;
  }
}
```

Ejemplo de uso en caso 2: P19/UseSuper.java

```
// Utilización de super para evitar la ocultación de nombres
class A {
  int i;
}

class B extends A {
  int i; // esta i oculta la i de A

  B(int a, int b) {
    super.i = a; // i in A
    i = b; // i in B
  }

  void show() {
    System.out.println("i en la superclase: " + super.i);
    System.out.println("i en la subclase: " + i);
  }
}

class UseSuper {
  public static void main(String args[]) {
    B subOb = new B(1, 2);

    subOb.show();
  }
}
```

7.3. Orden de ejecución de constructores

Los constructores de una jerarquía de clases se ejecutan en el orden de derivación.

Ejemplo: P20/CallingCons.java

```
// Muestra cuando se ejecutan los constructores.
class A {
  A() {
    System.out.println("En el constructor de A.");
  }
}
class B extends A {
  B() {
    System.out.println("En el constructor de B.");
  }
}
class C extends B {
  C() {
    System.out.println("En el constructor de C.");
  }
}
class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

La salida del programa es:

```
En el constructor de A.
En el constructor de B.
En el constructor de C.
```

7.4. Sobreescritura de métodos (Overriding)

Consiste en construir un método en una subclase con el mismo nombre, parámetros y tipo que otro método de una de sus superclases (inmediata o no).

Ejemplo: P21/Override.java

```
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }
  void show() {
    System.out.println("i y j: " + i + " " + j);
  }
}
class B extends A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  // muestra k -- sobreescribe el metodo show() de A
  void show() {
    System.out.println("k: " + k);
  }
}
class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);
    subOb.show(); // llama al metodo show() de B
  }
}
```

7.5. Selección de método dinámica

Es el mecanismo mediante el cual una llamada a una función sobreescrita se resuelve en **tiempo de ejecución** en lugar de en tiempo de compilación:
polimorfismo en tiempo de ejecución

> Cuando un método sobreescrito se llama a través de una referencia de la superclase, Java determina la versión del método que debe ejecutar en función del objeto que está siendo referenciado.

Ejemplo: P22/Dispatch.java

```
class A {
   void callme() {
     System.out.println("Llama al metodo callme dentro de A");
  }
}

class B extends A {
  void callme() {
    System.out.println("Llama al metodo callme dentro de B");
  }
}

class C extends A {
  void callme() {
    System.out.println("Llama al metodo callme dentro de C");
  }
}
```

```
class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // objeto de tipo A
    B b = new B(); // objeto de tipo B
    C c = new C(); // objeto de tipo C
    A r; // obtiene una referencia del tipo A
    r = a; // r hace referencia a un objeto A
    r.callme(); // llama al metodo callme() de A
    r = b; // r hace referencia a un objeto B
    r.callme(); // llama al metodo callme() de B
    r = c; // r hace referencia a un objeto C
    r.callme(); // llama al metodo callme() de C
  }
}
```

7.5.1. Aplicación de sobreescritura de métodos

Ejemplo: P23/FindAreas.java

```
class Figure {
  double dim1;
  double dim2;
  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }
  double area() {
    System.out.println("Area para Figure es indefinida.");
    return 0;
  }
}
```

```
class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }
  double area() {
    System.out.println("Dentro de Area para Rectangle.");
    return dim1 * dim2;
  }
}
class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }
  double area() {
    System.out.println("Dentro de Area para Triangle.");
    return dim1 * dim2 / 2;
  }
}
class FindAreas {
  public static void main(String args[]) {
    Figure f = new Figure(10, 10);
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref;
    figref = r;
    System.out.println("Area es " + figref.area());
    figref = t;
    System.out.println("Area es " + figref.area());
    figref = f;
    System.out.println("Area es " + figref.area());
  }
}
```

7.6. Clases abstractas

Permiten definir una superclase que define la estructura de las subclases, sin proporcionar una implementación completa de sus métodos.

- Todos los métodos abstractos (**abstract**) deben ser sobreescritos por las subclases.
- Los métodos abstractos tienen la forma:

  `abstract tipo nombre(parámetros);`
- Cualquier clase con uno o más métodos abstractos debe declararse como **abstract**
- No se pueden crear objetos de clases abstractas (usando **new**)
- No se pueden crear constructores **abstract** o métodos **static abstract**
- Si podemos declarar variables referencia de una clase abstracta

Ejemplo: P24/AbstractAreas.java

```
abstract class Figure {
  double dim1, dim2;
  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }
  abstract double area();
}
class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }
  double area() {
    System.out.println("Dentro del metodo area para un Rectangle.");
    return dim1 * dim2;
  }
}
```

```
class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }
  double area() {
    System.out.println("Dentro del metodo area para un Triangle.");
    return dim1 * dim2 / 2;
  }
}
class AbstractAreas {
  public static void main(String args[]) {
  // Figure f = new Figure(10, 10); // Esto no es correcto
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref; // esto es CORRECTO, no se crea ningún objeto
    figref = r;
    System.out.println("Area es " + figref.area());
    figref = t;
    System.out.println("Area es " + figref.area());
  }
}
```

7.7. Utilización de final con la herencia

La palabra reservada **final** tiene tres usos:

1. Para creación de constantes con nombre.
2. Para evitar sobreescritura de métodos

   Los métodos declarados como **final** no pueden ser sobreescritos
3. Para evitar la herencia

   Se usa **final** en la declaración de la clase para evitar que la clase sea heredada: O sea, todos sus métodos serán **final** implícitamente.

Ejemplo de uso en caso 2

```
class A {
  final void meth() {
    System.out.println("Este es un metodo final.");
  }
}

class B extends A {
  void meth() { // ERROR! No se puede sobreescribir.
    System.out.println("No es correcto!");
  }
}
```

Ejemplo de uso en caso 3

```
final class A {
  // ...
}

// La clase siguiente no es válida.
class B extends A { // ERROR! No se puede crear una subclase de A
  // ...
}
```

8. Paquetes e Interfaces

8.1. Paquetes

Un paquete es un contenedor de clases, que se usa para mantener el espacio de nombres de clase, dividido en compartimentos.

- Se almacenan de manera jerárquica: Java usa los directorios del sistema de archivos para almacenar los paquetes

  Ejemplo: Las clases del paquete **MiPaquete** (ficheros `.class` y `.java`) se almacenarán en directorio **MiPaquete**

- Se importan explícitamente en las definiciones de nuevas clases con la sentencia

  `import nombre-paquete;`

- Permiten restringir la visibilidad de las clases que contiene:

  Se pueden definir clases en un paquete sin que el mundo exterior sepa que están allí.

- Se pueden definir miembros de una clase, que sean sólo accesibles por miembros del mismo paquete

8.1.1. Definición de un paquete

Incluiremos la siguiente sentencia como primera sentencia del archivo fuente `.java`

`package nombre-paquete`

- Todas las clases de ese archivo serán de ese paquete
- Si no ponemos esta sentencia, las clases pertenecen al *paquete por defecto*
- Una misma sentencia `package` puede incluirse en varios archivos fuente
- Se puede crear una jerarquía de paquetes:
  `package paq1[.paq2[.paq3]];`
  Ejemplo: `java.awt.image`

8.1.2. La variable de entorno CLASSPATH

Supongamos que construimos la clase **PaquetePrueba** perteneciente al paquete **prueba**, dentro del fichero `PaquetePrueba.java` (directorio `prueba`)

- **Compilación**: Situarse en directorio `prueba` y ejecutar

  ```
  javac -bootclasspath $CLDC_PATH/common/api/classes
        -d tmpclasses prueba/PaquetePrueba.java
  ```

- **Preverificación**

  ```
  preverify -classpath $CLDC_PATH/common/api/classes:tmpclasses
        -d . prueba/PaquetePrueba
  ```

  o bien

  ```
  preverify -classpath $CLDC_PATH/common/api/classes:tmpclasses
        -d . tmpclasses
  ```

- **Ejecución**: Situarse en directorio padre de **prueba** y ejecutar

  ```
  kvm -classpath . prueba.PaquetePrueba
  ```

  o bien añadir directorio padre de **prueba** a **CLASSPATH** y ejecutar

  ```
  kvm prueba.PaquetePrueba
  ```

8.1.3. Ejemplo de paquete: P25/MyPack

```
package MyPack;

class Balance {
  String name;
  double bal;

  Balance(String n, double b) {
    name = n;
    bal = b;
  }
  void show() {
    if(bal<0)
      System.out.print("-->> ");
    System.out.println(name + ": $" + bal);
  }
}

class AccountBalance {
  public static void main(String args[]) {
    Balance current[] = new Balance[3];

    current[0] = new Balance("K. J. Fielding", 123.23);
    current[1] = new Balance("Will Tell", 157.02);
    current[2] = new Balance("Tom Jackson", -12.33);
    for(int i=0; i<3; i++) current[i].show();
  }
}
```

8.2. Protección de acceso

Las clases y los paquetes son dos medios de encapsular y contener el espacio de nombres y el ámbito de las variables y métodos.

- **Paquetes**: Actúan como recipientes de clases y otros paquetes subordinados.
- **Clases**: Actúan como recipientes de datos y código.

8.2.1. Tipos de acceso a miembros de una clase

Desde método en ...	**private**	**sin modif.**	**protected**	**public**
misma clase	sí	sí	sí	sí
subclase del mismo paquete	no	sí	sí	sí
no subclase del mismo paquete	no	sí	sí	sí
subclase de diferente paquete	no	no	sí	sí
no subclase de diferente paquete	no	no	no	sí

8.2.2. Tipos de acceso para una clase

- **Acceso por defecto**: Accesible sólo por código del mismo paquete
- **Acceso public**: Accesible por cualquier código

Ejemplo: P26 Ejemplo: P26

```
package p1;
public class Protection {
  int n = 1;
  private int n_pri = 2;
  protected int n_pro = 3;
  public int n_pub = 4;
  public Protection() {
    System.out.println("constructor base ");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
class Derived extends Protection {
  Derived() {
    System.out.println("constructor de Derived");
    System.out.println("n = " + n);
//  System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
class SamePackage {
  SamePackage() {
    Protection p = new Protection();
    System.out.println("constructor de SamePackage");
    System.out.println("n = " + p.n);
//  System.out.println("n_pri = " + p.n_pri);
    System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

```
package p2;

class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("constructor de Protection2");
//  System.out.println("n = " + n);
//  System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
  }
}
class OtherPackage {
  OtherPackage() {
    p1.Protection p = new p1.Protection();
    System.out.println("other package constructor");
//  System.out.println("n = " + p.n);
//  System.out.println("n_pri = " + p.n_pri);
//  System.out.println("n_pro = " + p.n_pro);
    System.out.println("n_pub = " + p.n_pub);
  }
}
```

8.3. Importar paquetes

Todas las clases estándar de Java están almacenadas en algún paquete con nombre. Para usar una de esas clases debemos usar su *nombre completo*. Por ejemplo para la clase **Date** usaríamos **java.util.Date**. Otra posibilidad es usar la sentencia **import**.

- **import**: Permite que se puedan ver ciertas clases o paquetes enteros sin tener que introducir la estructura de paquetes en que están incluidos, separados por puntos.
- Debe ir tras la sentencia **package**:

  ```
  import paquete[.paquete2].(nombre_clase|*);
  ```

- Al usar `*` especificamos que se importa el paquete completo. Esto incrementa el tiempo de compilación, pero no el de ejecución.
- Las clases estándar de Java están dentro del paquete **java**.
- Las funciones básicas del lenguaje se almacenan en el paquete. **java.lang**, el cual es importado por defecto.
- Al importar un paquete, sólo están disponibles los elementos públicos de ese paquete para clases que no sean subclases del código importado.

Ejemplo: P27

MyPack/Balance.java

```
package MyPack;
/* La clase Balance, su constructor, y su metodo show()
   deben ser públicos. Esto significa que pueden ser utilizados por
   código que no sea una subclase y esté fuera de su paquete.
*/
public class Balance {
  String name;
  double bal;
  public Balance(String n, double b) {
    name = n;
    bal = b;
  }

  public void show() {
    if(bal<0)
      System.out.print("-->> ");
    System.out.println(name + ": $" + bal);
  }
}
```

TestBalance.java

```
import MyPack.*;
class TestBalance {
  public static void main(String args[]) {
    /* Como Balance es pública, se puede utilizar la
       clase Balance y llamar a su constructor. */
    Balance test = new Balance("J. J. Jaspers", 99.88);
    test.show(); // también se puede llamar al metodo show()
  }
}
```

8.4. Interfaces

Son sintácticamente como las clases, pero no tienen variables de instancia y los métodos declarados no contienen cuerpo.

- Se utilizan para especificar lo que debe hacer una clase, pero no cómo lo hace.
- Una clase puede implementar cualquier número de interfaces.

8.4.1. Definición de una interfaz

Una interfaz se define casi igual que una clase:

```
acceso interface nombre {
  tipo_devuelto método1(lista_de_parámetros);
  tipo_devuelto método2(lista_de_parámetros);
  tipo var_final1=valor;
  tipo var_final2=valor;
// ...
  tipo_devuelto métodoN(lista_de_parámetros);
  tipo var_finalN=valor;
}
```

- *acceso* puede ser **public** o no usarse.
  - Si no se utiliza (acceso por defecto) la interfaz está sólo disponible para otros miembros del paquete en el que ha sido declarada.
  - Si se usa **public**, puede ser usada por cualquier código (todos los métodos y variables son implícitamente **públicos**).
- Los métodos de una interfaz son básicamente *métodos abstractos* (no tienen cuerpo de implementación).
- Un interfaz puede tener variables pero serán implícitamente **final** y **static**.

Ejemplo definición de interfaz

```
interface Callback {
  void callback(int param);
}
```

8.4.2. Implementación de una interfaz

- Para implementar una interfaz, la clase debe implementar todos los métodos de la interfaz. Se usa la palabra reservada **implements**.
- Una vez declarada la interfaz, puede ser implementada por varias clases.
- Cuando implementemos un método de una interfaz, tiene que declararse como **public**.
- Si una clase implementa dos interfaces que declaran el mismo método, entonces los clientes de cada interfaz usarán el mismo método.

Ejemplo de uso de interfaces: P28

```
class Client implements Callback {
  public void callback(int p) {
    System.out.println("callback llamado con " + p);
  }
  void nonIfaceMeth() {
    System.out.println("Las clases que implementan interfaces " +
                       "además pueden definir otros miembros.");
  }
}
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

8.4.3. Acceso a implementaciones a través de referencias de la interfaz

- Se pueden declarar variables usando como tipo un interfaz, para referenciar objetos de clases que implementan ese interfaz.
- El método al que se llamará con una variable así, se determina en tiempo de ejecución.

Ejemplo

```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

- Con estas variables sólo se pueden usar los métodos que hay en la interfaz.

Otro ejemplo: P29

```
// Otra implementación de  Callback.
class AnotherClient implements Callback {
  public void callback(int p) {
    System.out.println("Otra versión de callback");
    System.out.println("El cuadrado de p es " + (p*p));
  }
}


class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();
    c.callback(42);
    c = ob; // c hace referencia a un objeto AnotherClient
    c.callback(42);
  }
}
```

8.4.4. Implementación parcial

- Si una clase incluye una interfaz, pero no implementa todos sus métodos, entonces debe ser declarada como **abstract**.

8.4.5. Variables en interfaces

- Las variables se usan para importar constantes compartidas en múltiples clases.
- Si una interfaz no tiene ningún método, entonces cualquier clase que incluya esta interfaz no tendrá que implementar nada.

Ejemplo: P30/AskMe.java

```
import java.util.Random;
interface SharedConstants {
  int NO = 0;
  int YES = 1;
  int MAYBE = 2;
  int LATER = 3;
  int SOON = 4;
  int NEVER = 5;
}
class Question implements SharedConstants {
  Random rand = new Random();
  int ask() {
    int prob = (int) (100 * rand.nextDouble());
    if (prob < 30) return NO;          // 30%
    else if (prob < 60) return YES;    // 30%
    else if (prob < 75) return LATER; // 15%
    else if (prob < 98) return SOON;  // 13%
    else return NEVER;                 // 2%
  }
}
```

```
class AskMe implements SharedConstants {
  static void answer(int result) {
    switch(result) {
      case NO:
        System.out.println("No"); break;
      case YES:
        System.out.println("Si"); break;
      case MAYBE:
        System.out.println("Puede ser"); break;
      case LATER:
        System.out.println("Mas tarde"); break;
      case SOON:
        System.out.println("Pronto"); break;
      case NEVER:
        System.out.println("Nunca"); break;
    }
  }
  public static void main(String args[]) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
  }
}
```

8.4.6. Las interfaces se pueden extender

- Una interfaz puede heredar otra utilizando la palabra reservada **extends**.
- Una clase que implemente una interfaz que herede de otra, debe implementar todos los métodos de la cadena de herencia.

Ejemplo: P31/IFExtend.java

```
interface A {
  void meth1();
  void meth2();
}
interface B extends A {
  void meth3();
}
class MyClass implements B {
  public void meth1() {
    System.out.println("Implemento meth1().");
  }
  public void meth2() {
    System.out.println("Implemento meth2().");
  }
  public void meth3() {
    System.out.println("Implemento meth3().");
  }
}
class IFExtend {
  public static void main(String arg[]) {
    MyClass ob = new MyClass();
    ob.meth1();
    ob.meth2();
    ob.meth3();
  }
}
```

9. Gestión de excepciones

Una excepción es una condición anormal que surge en una secuencia de código durante la ejecución de un programa. O sea, es un error en tiempo de ejecución.

◊ **Excepciones y errores**

- CLDC incluye la mayoría de las excepciones definidas por el paquete `java.lang` de J2SE, pero la mayoría de las clases de error se han eliminado, dejando sólo las siguientes:
  - `java.lang.Error`
  - `java.lang.OutOfMemoryError`
  - `java.lang.VirtualMachineError`
- El método `Throwable.printStackTrace()` es parte de la especificación de CLDC (aunque no lo es la versión sobrecargada que redirige el trazado de la pila (stack) a otro sitio diferente al flujo de salida de error).

  Sin embargo el formato de salida de este método es dependiente de la implementación. En KVM este método sólo escribe el nombre de la excepción.

9.1. Fundamentos

- Cuando surge una condición excepcional se crea un objeto que representa la excepción, y se *envía* al método que ha provocado la excepción.

  Este método puede gestionar la excepción él mismo o bien pasarla al método llamante. En cualquier caso, la excepción es *capturada* y procesada en algún punto.
- La gestión de excepciones usa las palabras reservadas **try**, **catch**, **throw**, **throws** y **finally**.

Forma general de un bloque de gestión de excepciones

```
try {
  // bloque de código
}
catch (TipoExcepcion1 exOb){
  // gestor de excepciones para TipoExcepcion1
}
catch (TipoExcepcion2 exOb){
  // gestor de excepciones para TipoExcepcion2
}
// ...
finally {
  // bloque de código que se ejecutara antes de
  // que termine el bloque try
}
```

9.2. Tipos de excepción

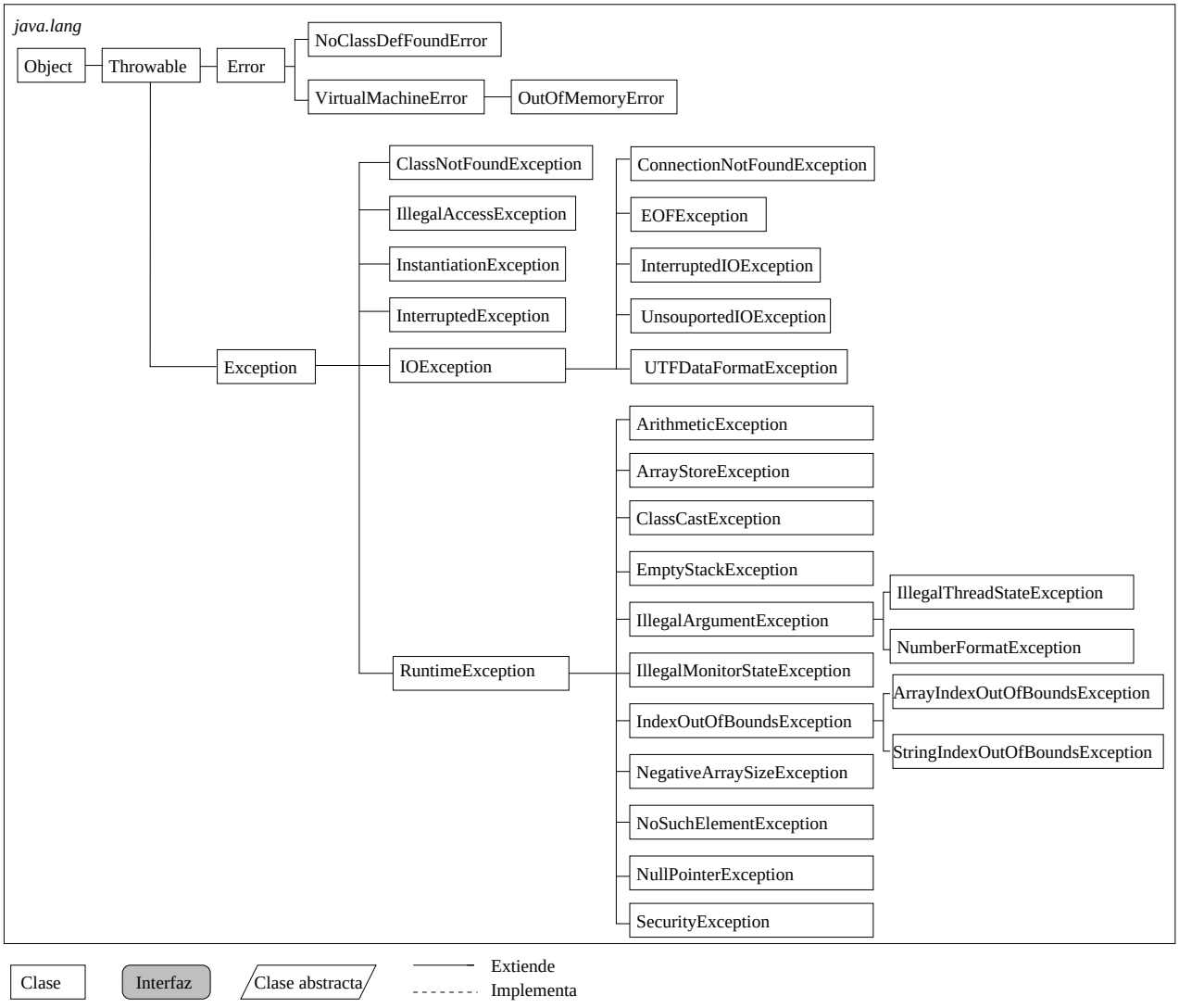
Todos los tipos de excepción son subclase de **Throwable**. Esta clase tiene dos subclases:

1. **Exception**: Se usa para las excepciones que deberían capturar los programas de usuario.

   Esta clase tiene como subclase a **RuntimeException**, que representa excepciones definidas automáticamente por los programas (división por 0, índice inválido de matriz, etc). Además tiene otras subclases como `ClassNotFoundException`, `InterruptedException`, etc.

2. **Error**: Excepciones que no se suelen capturar en condiciones normales.

   Suelen ser fallos catastróficos no gestionados por nuestros programas. Ejemplo: desbordamiento de la pila. En CLDC 1.1 hay dos subclases de `Error`: `VirtualMachineError` y `NoClassDefFoundError`.


java.lang
Object
Throwable
Error
NoClassDefFoundError
VirtualMachineError
OutOfMemoryError
Exception
ClassNotFoundException
IllegalAccessException
InstantiationException
InterruptedException
IOException
ConnectionNotFoundException
EOFException
InterruptedIOException
UnsouportedIOException
UTFDataFormatException
RuntimeException
ArithmeticException
ArrayStoreException
ClassCastException
EmptyStackException
IllegalArgumentException
IllegalThreadStateException
NumberFormatException
IllegalMonitorStateException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
NegativeArraySizeException
NoSuchElementException
NullPointerException
SecurityException
Clase
Interfaz
Clase abstracta
Extiende
Implementa

9.3. Excepciones no capturadas

Ejemplo: P32/Exc1.java

```
class Exc1 {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    System.out.println("Antes de Exc1.subroutine");
    Exc1.subroutine();
    System.out.println("Despues de Exc1.subroutine");
  }
}
```

- Cuando la máquina virtual Java detecta la división por 0 construye un objeto de excepción y lanza la excepción. Esto detiene la ejecución de `Exc1`, ya que no hemos incluido un *gestor de la excepción*.
- Cualquier excepción no tratada por el programa será tratada por el *gestor por defecto*.

  En el caso de J2SE, el gestor por defecto muestra la excepción y el trazado de la pila en la salida estándar, y termina el programa.

  Salida del programa

  ```
  Antes de Exc1.subroutine
  Exception in thread "main" java.lang.ArithmeticException: / by zero
          at Exc1.subroutine(Exc1.java:4)
          at Exc1.main(Exc1.java:8)
  ```

  En el caso de CLDC lo único que hace es terminar el programa.

  Salida del programa

  ```
  Antes de Exc1.subroutine
  ```

- En CLDC podemos usar la versión de depuración de la KVM (programa `kvm_g`) para obtener el trazado de la pila cuando se lanza la excepción. Necesitamos además hacer uso del método `Trowable.printStackTrace()` en la siguiente forma (P32/Exc1.version2.java):

```
try{
  int a = 10 / d;
}
catch (Exception ex){
  ex.printStackTrace();
}
```

Ahora la salida sería la siguiente:

```
Antes de Exc1.subroutine
java.lang.ArithmeticException
        at Exc1.subroutine(+5)
        at Exc1.main(+11)
Despues de Exc1.subroutine
```

9.4. try y catch

Si incluimos nuestro propio gestor de excepciones evitamos que el programa termine automáticamente. Usaremos un bloque **try-catch**.

Ejemplo: P33/Exc2.java

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // controla un bloque de código.
      d = 0;
      a = 42 / d;
      System.out.println("Esto no se imprimirá.");
    }
    catch (ArithmeticException e) {// captura el error de división
      System.out.println("División por cero.");
    }
    System.out.println("Después de la sentencia catch.");
  }
}
```

El objetivo de una sentencia **catch** bien diseñada debería ser resolver la condición de excepción y continuar.

Otro ejemplo: P34/HandleError.java

```
// Gestiona una excepción y continua.
import java.util.Random;
class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();
    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division por cero.");
        a = 0; // asigna a la variable el valor 0 y continua
      }
      System.out.println("a: " + a);
    }
  }
}
```

9.4.1. Descripción de una excepción

La clase **Throwable** sobreescribe el método **toString()** de la clase **Object**, devolviendo una cadena con la descripción de la excepción.

```
catch (ArithmeticException e) {
  System.out.println("Excepcion: " + e);
  a = 0; // hacer a=0 y continuar
}
```

Salida producida cuando se produce la excepción

```
Excepcion: java.lang.ArithmeticException
```

9.5. Cláusula catch múltiple

En algunos casos un bloque de código puede activar más de un tipo de excepción. Usaremos varios bloques **catch**.

El siguiente programa produce una excepción si se ejecuta sin parámetros y otra distinta si se ejecuta con un parámetro.

Ejemplo: P35/MultiCatch.java

```
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Division por 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Indice fuera de limites: " + e);
    }
    System.out.println("Despues del bloque try/catch.");
  }
}
```

Al ordenar los bloques **catch**, las subclases de excepción deben ir antes que la superclase (en caso contrario no se ejecutarían nunca y daría error de compilación por código no alcanzable).

Ejemplo: P36/SuperSubCatch.java

```
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    }
    catch(Exception e) {
      System.out.println("catch para cualquier tipo de excepción.");
    }
    /* Este catch nunca se ejecutará */
    catch(ArithmeticException e) { // ERROR - no alcanzable
      System.out.println("Esto nunca se ejecutará.");
    }
  }
}
```

9.6. Sentencias try anidadas

- Una sentencia **try** puede estar incluida dentro del bloque de otra sentencia **try**.
- Cuando un **try** no tiene **catch** para una excepción se busca si lo hay en el **try** más externo, y así sucesivamente.

Ejemplo: P37/NestTry.java

```
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;
      /* Si no hay ningún argumento en la línea de órdenes
         se generará una excepción de división por cero. */
      int b = 42 / a;
      System.out.println("a = " + a);
      try { // bloque try anidado
        /* Si se utiliza un argumento en la línea de órdenes
           se generará una excepción de división por cero. */
        if(a==1) a = a/(a-a); // división por cero
        /* Si se le pasan dos argumentos en la línea de órdenes,
           se genera una excepción al sobrepasar los límites
           del tamaño de la matriz. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // genera una excepción de fuera de límites
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Indice fuera de limites: " + e);
      }
    } catch(ArithmeticException e) {
      System.out.println("División por 0: " + e);
    }
  }
}
```

◊ Sentencias try anidadas en forma menos obvia

Ejemplo: P38/MethNestTry.java

```
/* Las sentencias try pueden estar implícitamente anidadas
   a través de llamadas a métodos. */
class MethNestTry {
  static void nesttry(int a) {
    try { // bloque try anidado
      /* Si se utiliza un argumento en la línea de órdenes, la
         siguiente sentencia efectúa división por cero */
      if(a==1) a = a/(a-a); // división por zero
      /* Si se le pasan dos argumentos en la línea de órdenes,
         se sobrepasan los límites de la matriz */
      if(a==2) {
        int c[] = { 1 };
        c[42] = 99; // genera una excepción de fuera de límites
      }
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Indice fuera de limites: " + e);
    }
  }
```

```
  public static void main(String args[]) {
    try {
      int a = args.length;
      /* Si no hay ningún argumento en la línea de órdenes, la
         siguiente sentencia generará una excepción de división
         por cero */
      int b = 42 / a;
      System.out.println("a = " + a);
      nesttry(a);
    } catch(ArithmeticException e) {
      System.out.println("División por 0: " + e);
    }
  }
}
```

9.7. Lanzar excepciones explícitamente: throw

Usando la sentencia **throw** es posible hacer que el programa lance una excepción de manera explícita: `throw objetoThrowable;`

El `objetoThrowable` puede obtenerse mediante:

1. El parámetro de una sentencia **catch**.
2. Con el operador **new** .

Ejemplo: P39/ThrowDemo.java

```
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Captura dentro de demoproc.");
      throw e; // relanza la excepción
    }
  }
```

```
  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Nueva captura: " + e);
    }
  }
}
```

El flujo de ejecución se detiene tras la sentencia **throw** (cualquier sentencia posterior no se ejecuta).

Salida del anterior ejemplo:

```
Captura dentro de demoproc.
Nueva captura: java.lang.NullPointerException: demo
```

9.8. Sentencia throws

Sirve para listar los tipos de excepción que un método puede lanzar.

- Debe usarse para proteger los métodos que usan a éste, si tal método lanza la excepción pero no la maneja.
- Es necesario usarla con todas las excepciones excepto **Error** y **RuntimeException** y sus subclases.
- Si las excepciones que lanza el método y no maneja no se ponen en **throws** se producirá error de compilación.

Forma general de declaración de método con throws

```
tipo metodo(lista_de_parametros) throws lista_de_excepciones
{
  // cuerpo del metodo
}
```

Ejemplo: P40/ThrowsDemo.java

```
// Programa erróneo que no compila
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Dentro de throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

El método que use al del **throws** debe capturar todas las excepciones listada con el **throws**.

Ejemplo: P41/ThrowsDemo.java

```
// Programa correcto
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Dentro de throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Captura " + e);
    }
  }
}
```

9.9. Sentencia finally

- Se puede usar esta sentencia en un bloque **try-catch** para ejecutar un bloque de código después de ejecutar las sentencias del **try** y del **catch**.
- Se ejecutará tanto si se lanza, como si no una excepción, y aunque no haya ningún **catch** que capture esa excepción.
- Podría usarse por ejemplo para cerrar los archivos abiertos.

Ejemplo de uso de finally: P42/FinallyDemo.java

```
class FinallyDemo {
  static void procA() {// Lanza una excepción fuera del metodo
    try {
      System.out.println("Dentro de procA");
      throw new RuntimeException("demo");
    } finally {
      System.out.println("Sentencia finally de procA");
    }
  }
  static void procB() {// Ejecuta la sentencia return
                       // dentro del try
    try {
      System.out.println("Dentro de procB");
      return;
    } finally {
      System.out.println("Sentencia finally de procB");
    }
  }
  static void procC() {// Ejecuta un bloque try normalmente
    try {
      System.out.println("Dentro de procC");
    } finally {
      System.out.println("Sentencia finally de procC");
    }
  }
```

```
  public static void main(String args[]) {
    try {procA();
    } catch (Exception e) {
      System.out.println("Excepción capturada");
    }
    procB(); procC();
  }
}
```

Salida del anterior programa

```
Dentro de procA
Sentencia finally de procA
Excepción capturada
Dentro de procB
Sentencia finally de procB
Dentro de procC
Sentencia finally de procC
```

9.10. Subclases de excepciones propias

- Sirven para crear tipos propios de excepción que permitan tratar situaciones específicas en una aplicación.
- Para ello sólo hay que definir una subclase de **Exception**.
- Estas subclases de excepciones no necesitan implementar nada.
- La clase **Exception** no define ningún método, sólo hereda los de **Throwable**.

Ejemplo: P43/ExceptionDemo.java

```
class MyException extends Exception {
  private int detail;
  MyException(int a) {
    detail = a;
  }
  public String toString() {
    return "MyException[" + detail + "]";
  }
}
class ExceptionDemo {
  static void compute(int a) throws MyException {
    System.out.println("Ejecuta compute(" + a + ")");
    if(a > 10)
      throw new MyException(a);
    System.out.println("Finalización normal");
  }
  public static void main(String args[]) {
    try {
      compute(1);
      compute(20);
    } catch (MyException e) {
      System.out.println("Captura " + e);
    }
  }
}
```

La salida de este programa sería:

```
Ejecuta compute(1)
Finalización normal
Ejecuta compute(20)
Captura MyException[20]
```

10. Programación Multihilo (Multihebra)

- Un programa multihilo contiene dos o más partes que pueden ejecutarse concurrentemente (aunque sólo tengamos una CPU).
- Esto permite escribir programas muy eficientes que utilizan al máximo la CPU, reduciendo al mínimo, el tiempo que está sin usarse.
- Java incluye características directamente en el lenguaje y API, para construir programas multihilo (a diferencia de otros lenguajes).

10.1. Hebras en CLDC

- Las máquinas virtuales de CLDC requieren proporcionar un entorno de programación multitarea incluso si el dispositivo no dispone de ella.
- Los mecanismos usados en J2SE para soportar la multitarea (palabra reservada `synchronized`, los métodos `Object.wait()`, `Object.notify()` y `Object.notifyAll()`, y la clase `Thread`) están incluidos en CLDC.

Algunas diferencias

- Sin embargo CLDC no soporta grupos de hebras o la clase `ThreadGroup`.
- El método `setName()` no existe. El método `getName()` existe sólo en CLDC 1.1
- Los métodos (deprecated) `resume()`, `suspend` y `stop()` se han eliminado.
- Los métodos `destroy()`, `interrupt()` y `isInterrupted()` no existen.
- El método `dumpStack()` se ha eliminado.

10.2. El hilo principal

- El hilo principal es lanzado siempre por la JVM (Java Virtual Machine) al ejecutar una aplicación.
- Desde este hilo se crearán el resto de hilos del programa.
- Debe ser el último que termine su ejecución. Cuando el hilo principal finaliza, termina el programa.

Ejemplo de acceso al hilo principal:
P44/CurrentThreadDemo.java

```
class CurrentThreadDemo {
  public static void main(String args[]) {
    Thread t = Thread.currentThread();
    System.out.println("Hilo actual: " + t);
    try {
      for(int n = 5; n > 0; n--) {
        System.out.println(n);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal");}
  }
}
```

Salida del programa

```
Hilo actual: Thread[Thread-0,5]
5
4
3
2
1
```

- En CLDC, cuando se usa `t` como argumento de `println()` aparece el nombre de la hebra y su prioridad.
- En CLDC no existe un método `setName(String)` en la clase `Thread` como en J2SE, lo que hace que sólo podamos asociar un nombre con el constructor `Thread(String)`.

10.3. Creación de un hilo

En Java hay dos formas de crear nuevos hilos:

- Implementando el interfaz Runnable.
- Extendiendo la clase Thread.

10.3.1. Implementación del interfaz Runnable

- Consiste en declarar una clase que implemente **Runnable** y sobreescribir el método `run()`:

  ```
  public abstract void run()
  ```

- Dentro de `run()` incluimos el código a ejecutar por el nuevo hilo.
- Luego se crea un objeto de la clase **Thread** dentro de esa clase. Al constructor de *Thread* le pasamos como argumento el objeto creado de la nueva clase (instancia de una clase que implemente **Runnable**):

  ```
  Thread(Runnable objetoHilo,String nombreHilo)
  ```

- Finalmente llamamos al método `start()` con el objeto anterior.

  ```
  synchronized void start()
  ```

Ejemplo: P45/ThreadDemo.java

```
class NewThread implements Runnable {
  Thread t;
  NewThread() {
    t = new Thread(this, "Hilo hijo");// Crea un nuevo hilo
    System.out.println("Hilo hijo: " + t);
    t.start(); // Comienza el hilo
  }
  public void run() {//Punto de entrada del segundo hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Hilo hijo: " + i);
        Thread.sleep(500); }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo hijo."); }
    System.out.println("Sale del hilo hijo.");
  }
}
class ThreadDemo {
  public static void main(String args[]) {
    new NewThread(); // crea un nuevo hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Hilo principal: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal.");
    }
    System.out.println("Sale del hilo principal.");
  }
}
```

Salida del anterior programa

```
Hilo hijo: Thread{Hilo hijo,5]
Hilo principal:5
Hilo hijo:5
Hilo hijo:4
Hilo principal:4
Hilo hijo:3
Hilo hijo:2
Hilo principal:3
Hilo hijo:1
Sale del hilo hijo.
Hilo principal:2
Hilo principal:1
Sale del hilo principal.
```

10.3.2. Extensión de la clase Thread

- Consiste en declarar una clase que herede de la clase **Thread** y sobreescribir también el método `run()`:
- Dentro de `run()` incluimos el código a ejecutar por el nuevo hilo.
- Luego se crea un objeto de la nueva clase.
- Finalmente llamamos al método `start()` con el objeto anterior.

Ejemplo: P46/ExtendThread.java

```
class NewThread extends Thread {
  NewThread() {
    super("Hilo Demo"); // Crea un nuevo hilo
    System.out.println("Hilo hijo: " + this);
    start(); // Comienza el hilo
  }
  public void run() {// Este es el punto de entrada del segundo hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Hilo hijo: " + i);
        Thread.sleep(500); }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo hijo.");
    }
    System.out.println("Sale del hilo hijo.");
  }
}
class ExtendThread {
  public static void main(String args[]) {
    new NewThread(); // crea un nuevo hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Hilo principal: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal.");
    }
    System.out.println("Sale del hilo principal.");
  }
}
```

10.3.3. Elección de una de las dos opciones

Depende del tipo de clase que se vaya a crear.

- Si la clase hereda de otra, no queda más remedio que implementar **Runnable**.
- Normalmente es más sencillo heredar de **Thread**.
- Pero algunos programadores piensan que una clase no debe extenderse si no va a ser ampliada o modificada. Por eso, si no vamos a sobreescribir ningún otro método de **Thread**, quizás sería mejor implementar **Runnable**.

10.4. Creación de múltiples hilos

Un programa en Java puede crear tantos hilos como quiera.

Ejemplo de creación de tres hilos: P47/MultiThreadDemo.java

```
class NewThread implements Runnable {
  String name; // nombre del hilo
  Thread t;
  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Nuevo hilo: " + t);
    t.start(); // Comienza el hilo
  }
  // Este es el punto de entrada del hilo.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    }
```

```
    catch (InterruptedException e) {
      System.out.println(name + "Interrupción del hilo hijo" +name);
    }
    System.out.println("Sale del hilo hijo" + name);
  }
}
class MultiThreadDemo {
  public static void main(String args[]) {
    new NewThread("Uno"); // comienzan los hilos
    new NewThread("Dos");
    new NewThread("Tres");
    try {
      // espera un tiempo para que terminen los otros hilos
      Thread.sleep(10000);
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal");
    }
    System.out.println("Sale del hilo principal.");
  }
}
```

Salida del programa

```
Nuevo hilo: Thread[Uno,5]
Nuevo hilo: Thread[Dos,5]
Nuevo hilo: Thread[Tres,5]
Uno: 5
Dos: 5
Tres: 5
Uno: 4
Dos: 4
Tres: 4
Uno: 3
Dos: 3
Tres: 3
Uno: 2
```

```
Dos: 2
Tres: 2
Uno: 1
Dos: 1
Tres: 1
Sale del hilo.Uno
Sale del hilo.Dos
Sale del hilo.Tres
Sale del hilo principal.
```

10.5. Utilización de isAlive() y join()

Hay dos formas de determinar si un método ha terminado.

- Con el método `isAlive()`. Devuelve **true** si el hilo al que se hace referencia está todavía ejecutándose.

  ```
  final boolean isAlive() throws InterruptedException
  ```

- Con el método `join()`. Este método detiene el hilo actual hasta que termine el hilo sobre el que se llama `join()`. Es usado por tanto para que unos hilos esperen a la finalización de otros.

  ```
  final void join throws InterruptedException
  ```

Ejemplo de uso de isAlive() y join(): P48/DemoJoin.java

```
class NewThread implements Runnable {
  String name; // nombre del hilo
  Thread t;
  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Nuevo hilo: " + t);
    t.start(); // Comienza el hilo
  }
```

```
  public void run() {// Este es el punto de entrada del hilo
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000); }
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo"+name); }
    System.out.println("Sale del hilo " + name);
  }
}
class DemoJoin {
  public static void main(String args[]) {
    NewThread ob1 = new NewThread("Uno");
    NewThread ob2 = new NewThread("Dos");
    NewThread ob3 = new NewThread("Tres");
    System.out.println("El hilo Uno está vivo: " + ob1.t.isAlive());
    System.out.println("El hilo Dos está vivo: " + ob2.t.isAlive());
    System.out.println("El hilo Tres está vivo: " + ob3.t.isAlive());
    try {// espera hasta que terminen los otros hilos
      System.out.println("Espera finalización de los otros hilos.");
      ob1.t.join(); ob2.t.join(); ob3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Interrupción del hilo principal"); }
    System.out.println("El hilo Uno está vivo: " + ob1.t.isAlive());
    System.out.println("El hilo Dos está vivo " + ob2.t.isAlive());
    System.out.println("El hilo Tres está vivo: " + ob3.t.isAlive());
    System.out.println("Sale del hilo principal.");
  }
}
```

Salida del programa

```
Nuevo hilo: Thread[Uno,5]
Nuevo hilo: Thread[Dos,5]
Nuevo hilo: Thread[Tres,5]
El hilo Uno está vivo: true
El hilo Dos está vivo: true
El hilo Tres está vivo: true
Espera finalización de los otros hilos.
Uno: 5
Dos: 5
Tres: 5
Uno: 4
Dos: 4
Tres: 4
Uno: 3
Dos: 3
Tres: 3
Uno: 2
Dos: 2
Tres: 2
Uno: 1
Dos: 1
Tres: 1
Sale del hilo Uno
Sale del hilo Dos
Sale del hilo Tres
El hilo Uno está vivo: false
El hilo Dos está vivo false
El hilo Tres está vivo: false
Sale del hilo principal.
```

10.6. Prioridades de los hilos

- La prioridad de un hilo es un valor entero que indica la prioridad relativa de un hilo respecto a otro.
- Se utiliza para decidir cuando pasar a ejecutar otro hilo (cambio de contexto).
  - Cuando un hilo cede el control (por abandono explícito o por bloqueo en E/S), se ejecuta a continuación el que tenga *mayor prioridad*.
  - Un hilo puede ser desalojado por otro con prioridad más alta, tan pronto como éste desee hacerlo.
- Pero en la práctica la cantidad de CPU que recibe cada hilo depende además de otros factores como la forma en que el sistema operativo implementa la multitarea.
- Para establecer la prioridad de un hilo usamos **setPriority()** de la clase **Thread** .

  `final void setPriority(int level)`
- `level` puede variar entre `MIN_PRIORITY` y `MAX_PRIORITY` (1 y 10 en la actualidad). La prioridad por defecto es `NORM_PRIORITY` (5 actualmente).
- Para obtener la prioridad de un hilo:

  `final int getPriority()`

Ejemplo: P49/HiLoPri.java

```
class clicker implements Runnable {
  int click = 0;
  Thread t;
  private volatile boolean running = true;
  public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
  }
```

```
    public void run() {
      while (running) {
        click++;
      }
    }
    public void stop() {
      running = false;
    }
    public void start() {
      t.start();
    }
  }
  class HiLoPri {
    public static void main(String args[]) {
      Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
      clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
      clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
      lo.start();
      hi.start();
      try {
        Thread.sleep(10000);
      } catch (InterruptedException e) {
          System.out.println("Hilo principal interrumpido.");}
      lo.stop();
      hi.stop();
      try {
        hi.t.join();
        lo.t.join();
      } catch (InterruptedException e) {
        System.out.println("InterruptedException capturada");
      }
      System.out.println("Hilo de prioridad baja: " + lo.click);
      System.out.println("Hilo de prioridad alta: " + hi.click);
    }
  }
```

Salida del programa en linux Redhat 8.0

```
Hilo de prioridad baja: 9636208
Hilo de prioridad alta: 22480211
```

10.7. Sincronización

- Cuando dos o más hilos necesitan acceder a un recurso compartido, entonces necesitan alguna forma de asegurar que el recurso se usa sólo por un hilo al mismo tiempo: **Sincronización**.
- Un *monitor* (*semáforo*) es un objeto usado como un cerrojo de exclusión mutua. Sólo un hilo puede poseer el monitor en un momento determinado.
- Cuando un hilo entra en el monitor, los demás hilos que intentan entrar en él, quedan suspendidos hasta que el primer hilo lo deja.
- En Java el código puede sincronizarse de dos formas:
  - Con métodos sincronizados.
  - Con sentencia **synchronized**.

10.7.1. Uso de métodos sincronizados

- Todos los objetos de Java tienen asociado su propio monitor implícito.
- Para entrar en el monitor de un objeto sólo hay que llamar a un método **synchronized**.
- Cuando un hilo esté ejecutando un método sincronizado, todos los demás hilos que intenten ejecutar cualquier método sincronizado del mismo objeto tendrán que esperar.

Ejemplo de programa que no usa sincronización: P50/Synch.java

```
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Interrumpido");
    }
    System.out.println("]");
  }
}
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    target.call(msg);
  }
}
```

```
class Synch {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hola");
    Caller ob2 = new Caller(target, "Mundo");
    Caller ob3 = new Caller(target, "Sincronizado");
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrumpido");
    }
  }
}
```

Salida del anterior programa

```
[Hola[Mundo[Sincronizado]
]
]
```

- El resultado de este programa muestra que la salida de los tres mensajes aparece mezclada, ya que los tres hilos están ejecutando el método `call()` del mismo objeto a la vez.
- Solución: Restringimos el acceso a `call()` a un sólo hilo en un momento determinado.

  Ejemplo: P51/Synch.java

  ```
  class Callme {
      synchronized void call(String msg) {
      ...
  ```

- Nueva salida:

  ```
  [Hola]
  [Mundo]
  [Sincronizado]
  ```

10.7.2. Sentencia synchronized

- A veces la solución de sincronizar un método no es posible porque no tenemos el código fuente de la clase, y no podemos hacer que el método sea sincronizado.
- En ese caso podemos usar la sentencia **synchronized**

```
synchronized(objeto){
  // sentencias que se sincronizan
}
```

- `objeto` es el objeto que sincronizamos.
- Esta sentencia hace que cualquier llamada a algún método de `objeto` se ejecutará después de que el hilo actual entre en el monitor de `objeto`.

Ejemplo que hace lo mismo de antes: P52/Synch1.java

```
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrumpido");
    }
    System.out.println("]");
  }
}
class Caller implements Runnable {
  String msg;
  Callme target;
  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this); t.start();
  }
```

```
  public void run() {
    synchronized(target) { target.call(msg);}
  }
}
class Synch1 {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hola");
    Caller ob2 = new Caller(target, "Sincronizado");
    Caller ob3 = new Caller(target, "Mundo");
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrumpido");
    }
  }
}
```

10.8. Comunicación entre hilos

- Los hilos pueden comunicarse entre sí mediante el uso de los métodos **wait()**, **notify()** y **notifyAll()** de la clase **Object** (son métodos finales).
  - **wait()** dice al hilo llamante que deje el monitor y que pase a estado suspendido (dormido) hasta que otro hilo entre en el mismo monitor y llame a **notify()**.
  - **notify()** despierta el primer hilo que llamó a **wait()** sobre el mismo objeto.
  - **notifyAll()** despierta todos los hilos que llamaron a **wait()** sobre el mismo objeto.
- Estos métodos sólo pueden ser llamados desde métodos sincronizados.

Productor/consumidor de un sólo carácter (versión errónea): P53/PC.java

```
class Q {
  int n;
  synchronized int get() {
    System.out.println("Obtengo: " + n);
    return n;
  }
  synchronized void put(int n) {
    this.n = n;
    System.out.println("Pongo: " + n);
  }
}
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
    this.q = q;
    new Thread(this, "Productor").start();
  }
  public void run() {
    int i = 0;
    while(true) {
      q.put(i++);
    }
  }
}
```

```
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumidor").start();
  }
  public void run() {
    while(true) {
      q.get();
    }
  }
}
class PC {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("Pulsa Control-C para parar.");
  }
}
```

Salida del programa

```
Pongo: 1
Obtengo: 1
Obtengo: 1
Obtengo: 1
Obtengo: 1
Obtengo: 1
Pongo: 2
Pongo: 3
Pongo: 4
Pongo: 5
Pongo: 6
Pongo: 7
Obtengo: 7
```

Solución correcta con wait y notify: P54/PCFixed.java

```
class Q {
  int n;
  boolean valueSet = false;
  synchronized int get() {
    if(!valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException capturada");
      }
    System.out.println("Obtengo: " + n);
    valueSet = false;
    notify();
    return n;
  }
  synchronized void put(int n) {
    if(valueSet)
      try {
        wait();
      } catch(InterruptedException e) {
        System.out.println("InterruptedException capturada");
      }
    this.n = n;
    valueSet = true;
    System.out.println("Pongo: " + n);
    notify();
  }
}
class Producer implements Runnable {
  Q q;
  Producer(Q q) {
    this.q = q;
    new Thread(this, "Productor").start();
  }
```

```
  public void run() {
    int i = 0;
    while(true) {
      q.put(i++);
    }
  }
}
class Consumer implements Runnable {
  Q q;
  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumidor").start();
  }
  public void run() {
    while(true) {
      q.get();
    }
  }
}
class PCFixed {
  public static void main(String args[]) {
    Q q = new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("Pulsa Control-C para parar.");
  }
}
```

Salida del programa

```
Pongo: 1
Obtengo: 1
Pongo: 2
Obtengo: 2
Pongo: 3
Obtengo: 3
```

```
Pongo: 4
Obtengo: 4
Pongo: 5
Obtengo: 5
Pongo: 6
Obtengo: 6
Pongo: 7
Obtengo: 7
```

10.8.1. Interbloqueos

- Un interbloqueo ocurre cuando dos hilos tienen una dependencia circular sobre un par de objetos sincronizados.
- Por ejemplo, un hilo entra en el monitor del objeto X y otro entra en el monitor de Y. Si X intenta llamar cualquier método sincronizado de Y, se bloqueará. Sin embargo, si el hilo de Y, intenta ahora llamar cualquier método sincronizado de X, los dos hilos quedarán detenidos para siempre.

Ejemplo de interbloqueo: P55/Deadlock.java

```
class A {
  synchronized void foo(B b) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " entró en A.foo");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("A Interrumpido");
    }
    System.out.println(name + " intentando llamar a B.last()");
    b.last();
  }
  synchronized void last() {
    System.out.println("Dentro de A.last");
  }
}
class B {
  synchronized void bar(A a) {
    String name = Thread.currentThread().getName();
    System.out.println(name + " entró en B.bar");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("B Interrumpido");
    }
    System.out.println(name + " intentando llamar a A.last()");
    a.last();
  }
  synchronized void last() {
    System.out.println("Dentro de A.last");
  }
}
```

```
class Deadlock implements Runnable {
  A a = new A();
  B b = new B();
  Deadlock() {
    Thread t = new Thread(this, "RacingThread");
    t.start();
    a.foo(b);
    System.out.println("Regreso al hilo principal");
  }
  public void run() {
    b.bar(a);
    System.out.println("Regreso al otro hilo");
  }
  public static void main(String args[]) {
    new Deadlock();
  }
}
```

Salida del programa hasta que queda bloqueado

```
Thread-0 entró en A.foo
RacingThread entró en B.bar
Thread-0 intentando llamar a B.last()
RacingThread intentando llamar a A.last()
```

10.9. Suspender, reanudar y terminar hilos

- La suspensión de la ejecución de un hilo es útil en algunos casos.

  Por ejemplo, un hilo puede usarse para mostrar la hora actual. Si el usuario no desea verla en un momento determinado, entonces tal hilo debería suspenderse.

- Una vez suspendido un hilo, éste puede volver a reanudar su ejecución.
- Para poder suspender, reanudar y terminar la ejecución de un hilo usaremos una variable bandera que indica el estado de ejecución del hilo. Haremos uso también de los métodos `wait()` y `notify()` o `notifyAll()` de la clase `Object`.

Ejemplo de uso: P57/SuspendResume.java

```
class NewThread implements Runnable {
  String name; // nombre del hilo
  Thread t;
  boolean suspendFlag;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Nuevo hilo: " + t);
    suspendFlag = false;
    t.start(); // Comienza el hilo
  }
```

```
    // Este es el punto de entrada del hilo.
    public void run() {
      try {
        for(int i = 15; i > 0; i--) {
          System.out.println(name + ": " + i);
          Thread.sleep(200);
          synchronized(this) {
            while(suspendFlag) {
              wait();
            }
          }
        }
      } catch (InterruptedException e) {
        System.out.println("Interrupción del hilo" + name);
      }
      System.out.println("Sale del hilo" + name);
    }
    void mysuspend() {
      suspendFlag = true;
    }
    synchronized void myresume() {
      suspendFlag = false;
      notify();
    }
  }
  class SuspendResume {
    public static void main(String args[]) {
      NewThread ob1 = new NewThread("Uno");
      NewThread ob2 = new NewThread("Dos");
      try {
        Thread.sleep(1000);
        ob1.mysuspend();
        System.out.println("Suspende el hilo Uno");
        Thread.sleep(1000);
        ob1.myresume();
```

```
        System.out.println("Reanuda el hilo Uno");
        ob2.mysuspend();
        System.out.println("Suspende el hilo Dos");
        Thread.sleep(1000);
        ob2.myresume();
        System.out.println("Reanuda el hilo Dos");
      } catch (InterruptedException e) {
        System.out.println("Interrupción del hilo principal");
      }
      // espera hasta que terminen los otros hilos
      try {
        System.out.println("Espera finalización de los otros hilos.");
        ob1.t.join();
        ob2.t.join();
      } catch (InterruptedException e) {
        System.out.println("Interrupción del hilo principal");
      }
      System.out.println("Sale del hilo principal.");
    }
  }
```

11. Connected Limited Device Configuration: CLDC

11.1. ¿Qué son las configuraciones en J2ME?

- Como ya se ha visto, una configuración define el entorno básico de ejecución a través de un conjunto mínimo de clases y una JVM (Java Virtual Machine) específica que se ejecuta en unos tipos especifícos de dispositivos.
- Los dos tipos de configuraciones existentes para J2ME son CLDC (para pequeños dispositivos) y CDC (para grandes dispositivos).

11.2. La configuración CLDC

- CLDC es el bloque básico en que se basan los *perfiles* de J2ME para pequeños dispositivos, tales como teléfonos móviles, PDAs (Personal Digital Assistants), Pocket PC, etc
- Estos dispositivos se caracterizan por tener **poca memoria y potencia de cálculo**, lo que les hace imposible ejecutar una plataforma Java completa.
- CLDC especifica un conjunto mínimo de paquetes y clases y una máquina virtual Java de funcionalidad reducida que puede ser instalada con las restricciones impuestas por tales dispositivos pequeños.

11.2.1. La máquina virtual Java de CLDC

- Las limitaciones hardware y software impuestas por los dispositivos a los que se dirige CLDC hacen impracticable soportar una máquina virtual Java completa o un conjunto completo de clases básicas.
- Por ejemplo, la ejecución de una simple aplicación "Hello, world"en una plataforma Windows requiere de alrededor de 16 MB de memoria.

- Sin embargo las limitaciones mínimas de memoria para la JVM de CLDC son:
  - 128 KB de ROM en versión CLDC 1.0 y 160 KB en 1.1, para almacenamiento persistente de la VM de Java y las librerías de clases de CLDC.
  - 32 KB o más de memoria volátil para alojarla en tiempo de ejecución (carga de clases y reserva en el heap y en la pila para objetos).
- CLDC define requerimientos mínimos en el propio lenguaje, en las librerías básicas y en la cantidad de memoria usada por la JVM.
- Además CLDC supone que el dispositivo huesped dispone de un sistema operativo que puede ejecutar y manejar la máquina virtual.
- Sin embargo CLDC no hace suposiciones sobre muchos otros temas. Por ejemplo:
  - No define el tipo de display.
  - No define el mecanismo de entrada tal como teclado o ratón.
  - No requiere de algún medio específico de almacenamiento para los datos de aplicación.
- Sin embargo los *perfiles* de J2ME, sí que dan requerimientos adicionales para el rango más limitado de dispositivos a los que se dirigen.

11.3. La librería de clases de CLDC

- El API de CLDC contiene un paquete específico de J2ME llamado `javax.microedition.io` y un pequeño subgrupo de paquetes del API de J2SE (Java 2 Standard Edition):
  - `java.io`
  - `java.lang`
  - `java.util`
- El API de CLDC no contiene todos los campos, métodos y clases definidos en los anteriores paquetes de J2SE (pueden estar excluidos los que sean deprecated (obsoletos) o por razones de limitación de recursos).

11.3.1. Paquete java.lang

- Proporciona clases fundamentales para el lenguaje Java. Contiene la mitad de las clases que hay en el mismo paquete de J2SE.
  - Clases `String` y `StringBuffer`: Ya comentadas anteriormente.
  - Clase `Thread` e interfaz `Runnable`: Ya comentadas anteriormente.
  - Clase `Math`: Contiene métodos y constantes estáticos usados para operaciones matemáticas.
  - Clase `Object`
  - Clases de envoltura para los tipos simples: `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`.
  - Las clases `System` y `Runtime`.
  - Clase `Class`.
  - Clase `Throwable`.
- Las omisiones más importantes son:
  - La mayoría de las clases `Error` (errores serios que los programas no suelen capturar) que no son necesarias debido a que la VM de CLDC no las soporta. Algunas de las clases de `Exception` también son omitidas.
  - En la versión CLDC 1.0 (pero sí en la 1.1) no están las clases `Float` y `Double`, ya que la VM no soporta operaciones en coma flotante.
  - Otras clases tales como `ClassLoader` `SecurityManager` y `StrictMath` que no son necesarias debido a que su utilidad no está en la especificación de CLDC.
  - Se eliminan los interfaces `Cloneable`, `Comparable` y `Serializable`.
- Algunas de las clases que hay incluidas no contienen implementaciones completas.

◊ La clase Object

- Esta clase es la clase base para todos los objetos de Java.
- Contiene métodos que serán comunes a todas las demás clases.

```
boolean equals(Object obj)
final Class getClass()
int hashCode()
final void notify()
final void notifyAll()
public final void wait(long timeout) throws InterruptedException
public String toString()
```

- Esta clase no tiene método `finalize()` debido a que la VM de CLDC no implementa la finalización. Tampoco contiene el método `clone()` (el interfaz `java.lang.Cloneable` tampoco está en CLDC).

Clases de envoltura para tipos simples

Existe una clase de *envoltura* por cada uno de los tipos simples.

- Eso es debido a que en Java los tipos simples no son objetos, lo que hace que no puedan ser utilizados en aquellos sitios dónde se espera que haya un objeto.
- Por ejemplo un `Vector` permite guardar una colección de objetos de cualquier tipo, pero deben ser objetos de alguna clase, y no tipos simples.

◊ Clases de envoltura para tipos numéricos

- Las operaciones en coma flotante no son soportadas por CLDC 1.0 pero sí por la 1.1.
- En CLDC no se incluye la clase `java.lang.Number`, con lo que las clases numéricas derivan directamente de `Object`.
- Tampoco existe el interfaz `java.lang.Comparable`, con lo que los números no pueden compararse directamente como en J2SE.
- En esta categoría se incluyen las clases `Byte Integer`, `Long`, `Short`. En CLDC 1.1 además tenemos `Float` y `Double`.
- Entre los métodos más usados tenemos:
  - Constructores: `Double(double valor)`, `Integer(integer valor)`, etc
  - En todas esta clases tenemos los métodos `double doubleValue()`, `float floatValue()`, `int intValue()`, etc.: Convierten el valor del número al tipo deseado.

◊ **Clase Character: envoltura para char**

- Constructor: `Character(char valor)`
- Método `char charValue()`: Devuelve el carácter que representa este objeto.
- Contiene métodos para ver si el valor del tipo simple es dígito, letra minúscula o mayúscula, etc.

```
static boolean isDigit(char ch)
static boolean isLowerCase(char ch)
static boolean isUpperCase(char ch)
```

- Métodos para convertir a minúscula o mayúscula:

```
static char toLowerCase(char ch)
static char toUpperCase(char ch)
```

◊ **Clase Boolean: envoltura para boolean**

- Constructor: `Boolean(boolean valor)`
- Esta clase contiene las constantes estáticas `TRUE` y `FALSE`.
- Método `boolean booleanValue()`: Devuelve el valor que represente este objeto.

◊ Características de reflexión

- La reflexión es una característica presente en J2SE que permite a un objeto consultar qué métodos y campos contiene él mismo.
- En CLDC no se está presente la reflexión con lo que todos los métodos de `java.lang.Class` relacionados con ella se han eliminado. Sólo se dispone en esta clase de los métodos `forName()` y `newInstance()`.

◊ Propiedades del sistema

- CLDC define sólo un conjunto pequeño de propiedades del sistema.

Propiedad	Significado	Ejemplo
`microedition.configuration`	Configuración J2ME soportada por la plataforma	CLDC-1.0
`microedition.encoding`	Codificación de caracteres soportada por la plataforma	ISO8859_1
`microedition.platform`	Plataforma o dispositivo	J2ME
`microedition.profile`	Perfil soportado por el dispositivo	MIDP-1.0

- El valor de una propiedad específica puede ser obtenido con el método `java.lang.System.getProperty()`;

```
String configuration = System.getProperty(
                                    "microedition.configuration");
```

- CLDC no incluye la clase `java.util.Properties` de J2SE, con lo que no se puede usar el método `getProperties()` para obtener una lista de todas las propiedades disponibles.

◊ Las clases System y Runtime

- Estas clases de J2SE contienen una colección de métodos que permiten operaciones a bajo nivel, tales como comenzar a ejecutar programas en código nativo desde la aplicación Java. Muchas de estas características se han eliminado:
  - Acceso directo a propiedades del sistema con `getProperties()`, `setProperty()` y `setProperties()`.
  - Métodos que permiten cambiar la fuente y destino de los flujos estándar de entrada, salida y error.
  - Métodos que proporcionan acceso a librerías de código nativo, ya que JNI (Java Native Interface) no está soportado.

11.3.2. Paquete java.util

Contiene clases de colección y clases relacionadas con la fecha y la hora, y la clase `Random` usada para generar números aleatorios.

◊ Clases de colección

CLDC incluye las siguientes clases e interfaz para manejar colecciones de objetos:

- Clase `Hashtable`
- Clase `Stack`
- Clase `Vector`
- Interfaz `Enumeration`

◊ Interfaz Enumeration

Define los métodos que permiten enumerar (obtener uno cada vez) los elementos de un conjunto de objetos. Esta interfaz define los dos métodos siguientes:

- `boolean hasMoreElements()` devuelve `true` mientras haya más elementos para extraer y `false` cuando se han enumerado todos los elementos.
- `Object nextElement()` devuelve el siguiente objeto de la enumeración como una referencia genérica a objeto, es decir, cada llamada a `nextElement()` obtiene el siguiente objeto de la enumeración.
- Algunas clases del API de Java utilizan enumeraciones.

Ejemplo: P60/EnumerateDemo.java

```
import java.util.Enumeration;
class Enum implements Enumeration {
  private int count = 0;
  private boolean more = true;
  public boolean hasMoreElements() {
    return more;
  }
  public Object nextElement() {
    count++;
    if(count > 4)
      more = false;
      return new Integer(count);
  }
}
class EnumerateDemo {
  public static void main(String args[]) {
    Enumeration enum = new Enum();

    while(enum.hasMoreElements()) {
      System.out.println(enum.nextElement());
    }
  }
}
```

Salida del programa

```
1
2
3
4
5
```

◊ Clase Vector

Los arrays o matrices de Java son de longitud fija. Una vez creados no se puede modificar su tamaño, lo que implica que necesitamos conocer a priori el número de elementos que van a almacenar.
Para resolver este problema, Java define la clase `Vector`.

- Un vector es una matriz de longitud variable de referencias a objetos.
- Los vectores se crean con un tamaño inicial y cuando se supera ese tamaño, automáticamente aumentan su tamaño.
- Al borrar objetos, el vector puede reducir su tamaño.
- Los constructores de esta clase son:

  ```
  Vector()
  Vector(int tamanoInicial)
  Vector(int tamanoInicial, int incrCapacidad)
  ```

- Los métodos más usados en esta clase son:
  - `final void addElement(Object elemento)`: Añade el objeto referenciado con `elemento` al vector.
  - `final Object elementAt(int posicion)`: Obtiene el elemento que se encuentra en la posición dada.
  - `final boolean contains(Object elemento)`: Permite determinar si el vector contiene un determinado elemento.
  - `final Object firstElement()`: Obtiene el primer elemento del vector.
  - `final Object lastElement()`: Obtiene el primer elemento del vector.
  - `final int indexOf(Object elemento)`: Devuelve la posición de la primera ocurrencia de elemento. Si el objeto no está en el vector devuelve -1.
  - `final int lastIndexOf(Object elemento)`: Devuelve la posición de la última ocurrencia de elemento.

- `final boolean removeElement(Object elemento)`: Elimina el elemento del vector. Si existe más de una ocurrencia del objeto, se elimina la primera. Devuelve true si se hace la operación correctamente y false si no encuentra el objeto.
- `final void removeElementAt(int posicion)`: Elimina el elemento de la posición especificada.
- `final Object elements()`: Devuelve una enumeración de los elementos del vector.

Ejemplo: P61/VectorDemo.java

```
import java.util.Vector;
import java.util.Enumeration;
class VectorDemo {
  public static void main(String args[]) {
    Vector v = new Vector(3, 2);
    System.out.println("Tamaño inicial: " + v.size());
    System.out.println("Capacidad inicial: " + v.capacity());
    v.addElement(new Integer(1));
    v.addElement(new Integer(2));
    v.addElement(new Integer(3));
    v.addElement(new Integer(4));
    System.out.println("Capacidad después de añadir 4 elementos: " +
                       v.capacity());
    v.addElement(new Double(5.45));
    System.out.println("Capacidad actual: " + v.capacity());
    v.addElement(new Double(6.08));
    v.addElement(new Integer(7));
    System.out.println("Capacidad actual: " +
                       v.capacity());
    v.addElement(new Float(9.4));
    v.addElement(new Integer(10));
    System.out.println("Capacidad actual: " +
                       v.capacity());
    v.addElement(new Integer(11));
    v.addElement(new Integer(12));
```

```
    System.out.println("Primer elemento: " +
                       (Integer)v.firstElement());
    System.out.println("Último elemento: " +
                       (Integer)v.lastElement());
    if(v.contains(new Integer(3)))
      System.out.println("El Vector contiene 3.");
    Enumeration vEnum = v.elements();
    System.out.println("\nElementos en el vector:");
    while(vEnum.hasMoreElements())
      System.out.print(vEnum.nextElement() + " ");
    System.out.println();
  }
}
```

Salida del programa

```
Tamaño inicial: 0
Capacidad inicial: 3
Capacidad después de añadir 4 elementos: 5
Capacidad actual: 5
Capacidad actual: 7
Capacidad actual: 9
Primer elemento: 1
Último elemento: 12
El Vector contiene 3.

Elementos en el vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

◊ Clase Stack

Es una subclase de la clase `Vector` que implementa una pila del tipo *último en entrar, primero en salir*

- `Stack()`: Constructor por defecto, que crea una pila vacía:
- Incluye todos los métodos definidos por `Vector` y añade además:
  - `Object push(Object elemento)`: Introduce el elemento en la pila, y además lo devuelve.
  - `Object pop()`: Devuelve y elimina el elemento superior de la pila. Este método lanza la excepción `EmptyStackException` si la pila está vacía.
  - `Object peek()`: Devuelve el elemento superior de la pila pero no lo borra.
  - `boolean empty()`: Devuelve true si la pila está vacía, y false en otro caso.
  - `int search(Object elemento)`: Determina si un objeto está en la pila y devuelve el número de operaciones pop que habría que realizar para que dicho elemento quede en la parte superior de la pila.

Ejemplo: P62/StackDemo.java

```
import java.util.Stack;
import java.util.EmptyStackException;
class StackDemo {
  static void showpush(Stack st, int a) {
    st.push(new Integer(a));
    System.out.println("push(" + a + ")");
    System.out.println("stack: " + st);
  }
  static void showpop(Stack st) {
    System.out.print("pop -> ");
    Integer a = (Integer) st.pop();
    System.out.println(a);
    System.out.println("stack: " + st);
  }
```

```
  public static void main(String args[]) {
    Stack st = new Stack();
    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);
    try {
      showpop(st);
    } catch (EmptyStackException e) {
      System.out.println("pila vacía");
    }
  }
}
```

Salida del programa

```
stack: []
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: []
pop -> pila vacía
```

◊ Clase Hashtable

Se usa para almacenar una colección de objetos que están indexados por cualquier otro objeto arbitrario.

- Una tabla hash almacena información utilizando un mecanismo llamado *hashing*: consiste en determinar un valor único (código hash) a partir del contenido de la clave.
- Tal código determina la posición en la que se almacenan los datos asociados a la clave.
- Cuando se utiliza una tabla hash hay que especificar el objeto que se utiliza como **clave** y el **valor**, es decir, los datos con los que se asocia esa clave.
- La transformación de la clave en un código hash se realiza automáticamente.
- Una tabla hash sólo puede almacenar objetos que sobreescriban los métodos `hashCode()` y `equals()` de la clase `Object`.
  - `int hashCode()`: Calcula y devuelve el código hash para el objeto.
  - `boolean equals(Object objeto)`: Compara si dos objetos son iguales.
- En CLDC tenemos dos constructores para esta clase:
  - `Hashtable()`
  - `Hashtable(int tamanoInicial)`

Ejemplo: P63/HTDemo.java

```
import java.util.Hashtable;
import java.util.Enumeration;
class HTDemo {
  public static void main(String args[]) {
    Hashtable balance = new Hashtable();
    Enumeration names, negbal;
    String str;
    double bal;
    balance.put("John Doe", new Double(3434.34));
    balance.put("Tom Smith", new Double(123.22));
    balance.put("Jane Baker", new Double(1378.00));
    balance.put("Tod Hall", new Double(99.22));
    balance.put("Ralph Smith", new Double(-19.08));
    names = balance.keys();
    while(names.hasMoreElements()) {
      str = (String) names.nextElement();
      System.out.println(str + ": " + balance.get(str));
    }
    System.out.println();
    bal = ((Double)balance.get("John Doe")).doubleValue();
    balance.put("John Doe", new Double(bal+1000));
    System.out.println("El nuevo saldo de John Doe es: " +
                       balance.get("John Doe"));
  }
}
```

Salida del programa

```
Tod Hall: 99.22
Ralph Smith: -19.08
John Doe: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22

El nuevo saldo de John Doe es: 4434.34
```

◊ La clase Date

- Un objeto `Date` es una envoltura para un número entero que representa una fecha y una hora como un desplazamiento (en milisegundos) desde 00:00 GMT (Greenwich Mean Time), 1 de Enero de 1970.
- Es una clase mucho más limitada que la correspondiente de J2SE.
- Sólo contiene constructores que crean un `Date` con el tiempo actual o un tiempo dado a partir de un desplazamiento desde la *época* (00:00 horas del 1 enero de 1970):
  - `Date()`
  - `Date(long desplazamiento)`
- Además contiene un par de métodos para definir el desplazamiento o recuperarlo, y un método `equals()` que compara un `Date` con otro.
  - `void setTime(long desplazamiento)`
  - `long getTime()`
  - `boolean equals(Object obj)`
- La versión CLDC 1.1 contiene también el método `toString()` que convierte el `Date` en un `String` en la forma
  `dow mon dd hh:mm:ss zzz yyyy`

  `public String toString()`
- Esta clase se suele usar junto con `Calendar` para convertir entre diferentes `Date`s y para transformar la representación del `Date` en algo más significativo que un desplazamiento.

◊ La clase TimeZone

- Un objeto `TimeZone` representa el desplazamiento de una zona respecto a la zona horaria GMT.
- Esta clase se usa debido a que todas las fechas en Java se representan con desplazamientos respecto a 00:00 GMT, 1 de Enero de 1970, con lo que se necesita conocer el desplazamiento local respecto a GMT para formatear la hora correspondiente a tu zona horaria actual.
- De nuevo la clase `TimeZone` de CLDC es mucho más pobre que la correspondiente de J2SE.
- Entre los métodos más importantes tenemos los métodos estáticos `getDefault()` y `getDefault(String)`, que obtienen la zona horaria por defecto o bien la especificada con el `String` para el dispositivo en cuestión.

```
static TimeZone getDefault()
static TimeZone getDefault(String ID)
```

◊ La clase Calendar

- Esta clase es de nuevo una versión simplificada de la correspondiente de J2SE.
- Su uso principal es convertir entre un instante de tiempo dado con un objeto `Date` y los correspondientes día, mes, año, hora, minutos y segundos.
- `Calendar` es una clase abstracta. Para obtener una instancia de ella, se pueden usar los métodos estáticos, que devuelven un objeto de una subclase de `Calendar` que ya implementan las reglas adecuadas al entorno concreto en que funciona el dispositivo:

  ```
  static Calendar getInstance()
  static Calendar getInstance(TimeZone zone)
  ```

- Los métodos más usados son los siguientes:
  - El método `setTime(Date)` permite establecer hora y fecha mediante un `Date`:

    ```
    void setTime(Date date)
    ```

  - El método `set()` establece un nuevo valor pero sólo para el campo seleccionado.

    ```
    void set(int field, int value)
    ```

  - El método `get()` permite obtener el valor de los distintos campos (hora, minutos, segundos, etc) usando constantes tales como `Calendar.HOUR`, `Calendar.MINUTE`, `Calendar.SECOND`, etc.

    ```
    int get(int field)
    ```

  - El método `String toString()` convierte el `Calendar` en un `String` en la forma `Tue, 9 Apr 2002 12:00:00 UTC`

Ejemplo

```
Calendar cal = Calendar.getInstance();
Date date = new Date();
cal.setTime(date);
int month = cal.get(Calendar.MONTH);
int day = cal.get(Calendar.DAY_OF_MONTH);
```

Otro ejemplo

```
// Obtenemos un Calendar y obtenemos los milisegundos
// de la fecha actual
Calendar cal = Calendar.getInstance();
Date date = new Date();
long offset = date.getTime();

// Sumamos 20 días a la fecha actual
final long MILLIS_PER_DAY = 24 * 60 * 60 * 1000L;
offset += 20 * MILLIS_PER_DAY;
date.setTime(offset);

// Instanciamos la nueva fecha en el offset del Calendar
cal.setTime(date);

// Obtenemos la fecha ya ajustada
month = cal.get(Calendar.MONTH);
day = cal.get(Calendar.DAY_OF_MONTH);
```

Otro ejemplo

```
// Obtenemos día y mes de fecha actual
Calendar cal = Calendar.getInstance();
Date date = new Date();
cal.setTime(date);
int month = cal.get(Calendar.MONTH);
int day = cal.get(Calendar.DAY_OF_MONTH);

// Sumamos 20 días al campo día en el Calendar
cal.set(Calendar.DAY_OF_MONTH, day+20);

// Obtenemos la fecha ya ajustada: NO FUNCIONA BIEN
month = cal.get(Calendar.MONTH);
day = cal.get(Calendar.DAY_OF_MONTH);
```

◊ La clase Random

Se usa para generar una secuencia de números seudoaleatorios.

- Constructores:

```
public Random();
public Random(long semilla);
```

- Métodos:
  - `protected int next(int bits)`
  - `public int nextInt()`: Genera el siguiente número aleatorio **int** uniformemente distribuido entre el conjunto de los números **int**.
  - `public int nextInt(int n)`: Genera el siguiente número aleatorio **int** uniformemente distribuido entre 0 y el números **int** pasado como argumento.
  - `public int nextLong()`: Genera el siguiente número aleatorio **long** uniformemente distribuido entre el conjunto de los números **long**.
  - `public float nextFloat()`: Genera el siguiente número aleatorio **float** uniformemente distribuido entre 0,0 y 1,0.
  - `public double nextDouble()`: Genera el siguiente número aleatorio **double** uniformemente distribuido entre 0,0 y 1,0.
  - `public void setSeed(long seed)`: Establece la semilla.

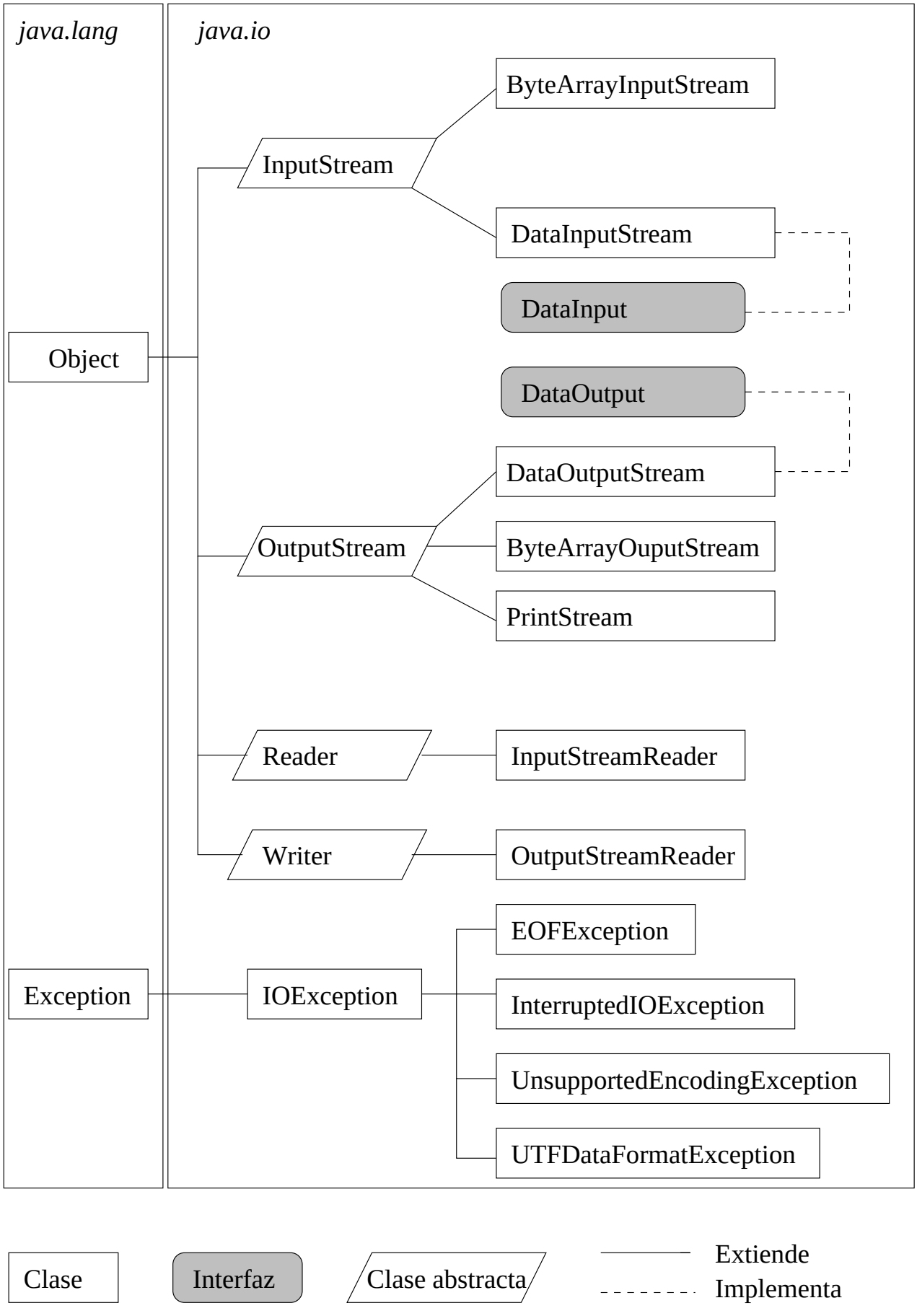
11.4. Entrada/salida en Java

11.4.1. Introducción

- Java realiza entradas y salidas a través de **streams** (flujos de datos).
- Un stream es una abstracción que produce o consume información.
- Un flujo está relacionado con un dispositivo físico a través del sistema de E/S de Java.
- Todos los flujos se comportan de la misma manera, incluso aunque estén relacionados con distintos dispositivos físicos.
- Un flujo puede abstraer distintos tipos de entrada: archivo de disco, teclado, conexión a red.
- Un flujo puede abstraer distintos tipos de salida: consola, archivo de disco o conexión a red.
- Con un stream la información se traslada **en serie** (caracter a caracter o byte a byte) a través de esta conexión.

11.4.2. Clases de CLDC para E/S de datos: Paquete java.io

- El paquete `java.io` contiene las clases necesarias para E/S en Java a través de flujos de datos (data streams).
- Este paquete de CLDC es un subconjunto del correspondiente de JDK.
- Dentro de este paquete existen dos familias de jerarquías distintas para la E/S de datos.
  - Clases que operan con bytes: Derivan de las clases `InputStream` o `OutputStream`.
  - Clases que operan con caracteres (un carácter en Java ocupa dos bytes porque sigue el código **Unicode**): Derivan de las clases abstractas `Reader` o `Writer`.


java.lang
java.io
ByteArrayInputStream
InputStream
DataInputStream
DataInput
Object
DataOutput
DataOutputStream
OutputStream
ByteArrayOuputStream
PrintStream
Reader
InputStreamReader
Writer
OutputStreamReader
EOFException
Exception
IOException
InterruptedIOException
UnsupportedEncodingException
UTFDataFormatException
Clase
Interfaz
Clase abstracta
Extiende
Implementa

11.4.3. Clases para flujos de bytes

Las clases `InputStream` y `OutputStream` son superclases abstractas de todos los flujos orientados a bytes, tal como en J2SE.

♢ Clase java.io.InputStream

Superclase abstracta de todos los flujos orientados a entrada de bytes. Los métodos de esta clase lanzan una `IOException` si se producen condiciones de error.

- Constructor

  `InputStream()`

- Métodos
  - `int available()`: Devuelve los bytes disponibles en el flujo.
  - `void close()`: Cierra el flujo y libera los recursos que usa.
  - `void mark(int readlimit)`: Establece una marca en la posición actual del flujo, que permite regresar posteriormente a esa posición.
  - `boolean markSupported()`: Indica si el flujo permite poner marcas.
  - `int read()`
    `int read(byte[] buffer)`
    `int read(byte[] buffer, int offset, int length)`: Lee bytes del flujo de entrada. Devuelve -1 si no hay bytes disponibles.
  - `void reset()`: Reposiciona el flujo en la marca establecida previamente.
  - `long skip(long bytecount)`: Salta `bytecount` bytes devolviendo el número de bytes saltados.

Ejemplo

```
try{
  InputStream is = Conector.openInputStream(
                              "socket://127.0.0.1:8888");
  int ch;
  while((ch = in.read()) >0) {
    // hacer algo con el dato leido
  }
} catch (IOException x) {
  // Manejar la excepción
}
```

♢ Clase java.io.OutputStream

Superclase de todos los flujos orientados a salida de bytes.
Los métodos de esta clase devuelven void y lanzan una `IOException` si se producen condiciones de error.

- Constructor.

  `OutputStream()`

- Métodos
  - `void close()`: Cierra el flujo y libera los recursos que utiliza.
  - `void flush()`: Fuerza a escribir los posibles bytes que haya almacenados en un buffer de memoria.
  - `void write(int b)`
    `void write(byte[] bytebuffer)`
    `void write(byte bytebuffer[], int offset, int count)`:
    Escribe un byte o un array de bytes en el flujo.

◊ Clase ByteArrayInputStream

Flujo de entrada que usa un array de bytes como origen de los datos.

- Es útil cuando tenemos los datos almacenados en un array de bytes y queremos leerlos como si proviniesen de un fichero, tubería o socket.
- Un ejemplo de su uso es con los `RecorStore`s de MIDP que se estudiarán más adelante. Estos son una especie de ficheros que guardan los datos en arrays de bytes. En los MIDLets se suelen leer los datos usando un `ByteArrayInputStream`
- Constructores de `ByteArrayInputStream`:

  ```
  ByteArrayInputStream(byte array[])
  ByteArrayInputStream(byte array[],int offset, int numBytes)
  ```

- Los métodos que contiene son los heredados de `InputStream`.

Ejemplo de uso de ByteArrayInputStream: P70/ByteArrayInputStreamDemo.java

```
import java.io.*;
class ByteArrayInputStreamDemo {
  public static void main(String args[]) throws IOException {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    byte b[];
    b=tmp.getBytes();
    ByteArrayInputStream input1 = new ByteArrayInputStream(b);
    ByteArrayInputStream input2 = new ByteArrayInputStream(b, 0, 3);
  }
}
```

Otro ejemplo de ByteArrayInputStream: P71/ByteArrayInputStreamReset.java

```
import java.io.*;
class ByteArrayInputStreamReset {
  public static void main(String args[]) throws IOException {
    byte b[] = { 'a', 'b', 'c' };
    ByteArrayInputStream in = new ByteArrayInputStream(b);
    for (int i=0; i<2; i++) {
      int c;
      while ((c = in.read()) != -1) {
        if (i == 0) {
          System.out.print((char) c);
        } else {
          System.out.print(Character.toUpperCase((char) c));
        }
      }
      System.out.println();
      in.reset();
    }
  }
}
```

Salida

```
abc
ABC
```

◊ **Clase ByteArrayOutputStream** Flujo de salida que usa un array de bytes como destino de los datos.

- Constructores de `ByteArrayOutputStream`:
  - `ByteArrayOutputStream()`: Crea un búfer con 32 bytes.
  - `ByteArrayOutputStream(int numBytes)`: Crea un búfer con el tamaño especificado.
- A parte de los métodos heredados de `OutputStream` contiene los siguientes:
  - `public void reset()`: Descarta todos los datos almacenados en el flujo, hasta el momento, al hacer igual a cero el campo `count` (el cual representa el número de bytes válidos en el vector dónde se almacenan los datos (bufer)).
  - `public int size()`: Devuelve el tamaño actual del búfer.
  - `public byte[] toByteArray()`: Devuelve una copia del búfer del flujo.
  - `public String toString()`: Convierte el contenido del búfer en un String.

Ejemplo: P72/ByteArrayOutputStream.java

```
import java.io.*;
class ByteArrayOutputStreamDemo {
  public static void main(String args[]) throws IOException {
    ByteArrayOutputStream f = new ByteArrayOutputStream();
    String s = "Esto se meterá en el array";
    byte buf[] ;
    buf=s.getBytes();
    f.write(buf);
    System.out.print("Se escribe el bufer como un string: ");
    System.out.println(f.toString());
    System.out.print("Se lee del buffer y guardamos en array: ");
    byte b[] = f.toByteArray();
    for (int i=0; i<b.length; i++) {
      System.out.print((char) b[i]);
    }
    System.out.println("\nHacemos un reset");
    f.reset();
    for (int i=0; i<3; i++) f.write('X');
    System.out.println("Volvemos a escribir el bufer"+
                       " como un string: " + f.toString());
  }
}
```

Salida del programa

```
Se escribe el bufer como un string: Esto se meterá en el array
Se lee del buffer y guardamos en array: Esto se meter? en el array
Hacemos un reset
Volvemos a escribir el bufer como un string: XXX
```

◊ Clase DataOutputStream

Esta subclase de `OutputStream` proporciona métodos para escribir tipos de datos básicos de Java de un forma independiente de la máquina, en el subyacente flujo de salida.

- Esta clase implementa los métodos del interfaz `DataOutput`.
- Las instancias de esta clase pueden crearse directamente con el constructor, o bien pueden ser obtenidas de otras formas, como por ejemplo con el método `openDataOutputStream()` de la clase `javax.microedition.io.Connector`.
- Constructor:

  ```
  public DataOutputStream(OutputStream in)
  ```

- A parte de los métodos heredados de `OutputStream` tenemos:
  - `void writeBoolean(boolean v)`: Escribe en el flujo el valor **boolean** usando un byte.
  - `void writeChar(int v)`: Escribe en el flujo el carácter usando dos bytes.
  - `void writeChars(String v)`: Escribe el **String** en el flujo como una secuencia de caracteres (dos bytes cada uno).
  - `void writeUTF(String s)`: Escribe el **String** en el flujo usando codificación UTF-8.
  - Escritura de enteros:
    - `void writeByte(int v)`: Escribe el entero usando un byte.
    - `void writeShort(int v)`: Escribe el entero (short) usando dos bytes.
    - `void writeInt(int v)`: Escribe el entero usando cuatro bytes.
    - `void writeLong(long v)`: Escribe el entero con ocho bytes.
  - Escritura de números en coma flotante (sólo en CLDC 1.1):
    - `void writeFloat(float v)`: Escribe el **float** usando cuatro bytes.
    - `void writeDouble(double v)`: Escribe el **double** usando ocho bytes.

Ejemplo de DataOutputStream usando un RecorStore

```
public class Record{
  public String nombreJugador;
  public int puntos;
}

Record record = new Record();
record.nombreJugador = "Tomas";
record.puntos = 12345678;
// Crear los flujos de salida
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream os = new DataOutputStream(baos);
// Escribir los valores en los flujos
os.writeUTF(record.nombreJugador);
os.writeInt(record.puntos);
os.close();
//Obtener el array de bytes con los valores guardados
byte[] data = baos.toByteArray();
//Escribir el record en el record store
int id = recordStore.addRecord(data,0,data.length);
```

◊ Clase DataInputStream

Esta subclase de `InputStream` proporciona métodos para leer tipos de datos básicos de Java de un forma independiente de la máquina, a partir del subyacente flujo de entrada.

- Esta clase implementa los métodos del interfaz `DataInput`.
- Las instancias de esta clase pueden crearse directamente con el constructor, o bien pueden ser obtenidas de otras formas, como por ejemplo con el método `openDataInputStream()` de la clase `javax.microedition.io.Connector`.
- Constructor:

  ```
  public DataInputStream(InputStream in)
  ```

- A parte de los métodos heredados de `InputStream` tenemos:
  - `boolean readBoolean()`: Lee un byte del flujo, devolviendo **true** si no es cero y **false** en otro caso.
  - `char readChar()`: Lee dos bytes del flujo, y los devuelve como char (en código Unicode).
  - `String readUTF()`: Lee bytes del flujo, y los devuelve como un `String` con representación Unicode. En el flujo se supone que el string está almacenado en formato UTF-8.
  - Lectura de enteros:
    - `byte readByte()`: Lee un byte del flujo, que es tratado como un entero con signo en el rango -128 a 127.
    - `int readUnsignedByte()`: Lee un byte, y lo interpreta como un entero con signo en el rango 0 a 255, devolviéndolo como un **int**.
    - `short readShort()`: Lee dos bytes, y los devuelve en un **short**.
    - `int readUnsignedShort()`: Lee dos bytes, y los interpreta como un short sin signo, devolviéndolo en un **int**.
    - `int readInt()`: Lee cuatro bytes, y los devuelve en un **int**.
    - `long readLong()`: Lee ocho bytes, y los devuelve en un **long**.

- Lectura de números en coma flotante (sólo en CLDC 1.1):
  - `float readFloat()`: Lee cuatro bytes, y los devuelve en un **float**.
  - `float readDouble()`: Lee ocho bytes, y los devuelve en un **double**.
- Otros métodos:
  - `void readFully(byte[] b)`: Lee parte de los bytes del flujo, guardándolos en el array pasado como parámetro. Se leen tantos bytes, como la longitud de tal array.
  - `void readFully(byte[] b, int off,int len)`: Similar al anterior.

Ejemplo de DataInputStream con el anterior RecordStore

```
byte[] data = recordStore.getRecord(recordId);
DataInputStream is = new DataInputStream(
                          new ByteArrayInputStream(data));
Record record = new Record();
record.nombreJugador = is.readUTF();
record.score = is.readInt();
is.close();
```

◊ Clase PrintStream

Extiende `OutputStream` y es usada para añadir funcionalidad a otro flujo de salida de bytes.
Esta funcionalidad es la capacidad de convertir los distintos tipos básicos de Java y objetos, en un formato que puede imprimirse en el subyacente `OutputStream` pasado al constructor de `PrintStream`.

- `System.out` y `System.err` son ejemplos de `PrintStream`.
- Constructor:

  ```
  public PrintStream(OutputStream out);
  ```

- Los métodos introducidos por esta clase son del tipo `print(tipo valor)` o `println(tipo valor)`, que convierten el valor en un `String` (codificado con la codificación usada por el dispositivo), y luego lo escriben en el flujo.

11.4.4. Clases para flujos de caracteres

◊ Clase Reader

Es una clase abstracta que proporciona soporte para leer flujos de caracteres.

- Un `Reader` se diferencia de un `InputStream` en que trabaja con caracteres Unicode de 16 bits en lugar de los 8 bits usados por los `InputStream`.
- Un flujo de entrada `InputStream` de 8 bits puede convertirse en una secuencia de caracteres Unicode usando el constructor de la subclase `InputStreamReader`.
- La ventaja fundamental del uso de un `Reader` es que la codificación de los caracteres se traduce automáticamente en la operación de lectura, desde la representación en bytes de los datos a la representación en caracteres de tales datos.
- Métodos:
  - `void close()`
  - `void mark(int readAheadLimit)`
  - `boolean markSupported()`
  - `void reset()`
  - `long skip(long n)`: Salta n caracteres.
  - `boolean ready()`: Indica si el flujo está preparado para poder leer de él.
  - `int read()`: Lee un caracter.
  - `int read(char[] cbuf)`: Lee un array de caracteres.
  - `int read(char[] cbuf,int off,int len)`: Similar al anterior.

◊ **Writer**

Es de nuevo una clase abstracta que define métodos implementados por las subclases para proporcionar salida de caracteres.

- Un `Writer` se diferencia de un `OutputStream` en que trabaja con caracteres Unicode de 16 bits en vez de con los 8 bits de los `OutputStream`.
- Una secuencia de caracteres Unicode puede convertirse en un flujo de datos de 8 bits usando un constructor de la subclase `OutputStreamWriter`.
- La ventaja fundamental del uso de un `Writer` es de nuevo que la codificación de los caracteres se translada automáticamente entre la representación en bytes de los datos y la representación en caracteres.
- Métodos:
  - `void close()`
  - `void flush()`
  - `void write(int c)`: Escribe un caracter.
  - `void write(char[] cbuf)`: Escribe un array de caracteres.
  - `void write(char[] cbuf,int off,int len)`: Similar al anterior.
  - `void write(String str)`: Escribe un String.
  - `void write(String str,int off,int len)`: Escribe un trozo del String.

◊ InputStreamReader

Esta clase extiende la clase `Reader` y proporciona una implementación para leer caracteres (8 bits) de un flujo de bytes `InputStream` convirtiéndolos en caracteres Unicode.

- O sea, los bytes leidos del flujo son traducidos automáticamente en caracteres.
- El mapping de los bytes leidos del `InputStream` a los caracteres que devuelve el `InputStreamReader` se lleva a cabo con la codificación que especifiquemos en el constructor, o bien con la codificación por defecto del dispositivo.
- Constructores:

  ```
  public InputStreamReader(InputStream is)
  InputStreamReader(InputStream is, String enc)
  ```

- Los métodos que tiene son los heredados y sobreescritos de `Reader`

◊ OutputStreamWriter

Extiende la clase `Writer` y proporciona la implementación necesaria para poder escribir caracteres Unicode en un flujo de bytes `OutputStream` (escribe 8 bits por cada carácter).

- O sea, los caracteres escritos en este flujo son traducidos automáticamente de Unicode a bytes.
- De nuevo, el mapping entre el codigo Unicode (de 16 bits) y los bytes requeridos por el subyacente `OutputStreamWriter` se lleva a cabo con la codificación que especifiquemos en el constructor, o bien con la codificación por defecto del dispositivo.
- Constructores:

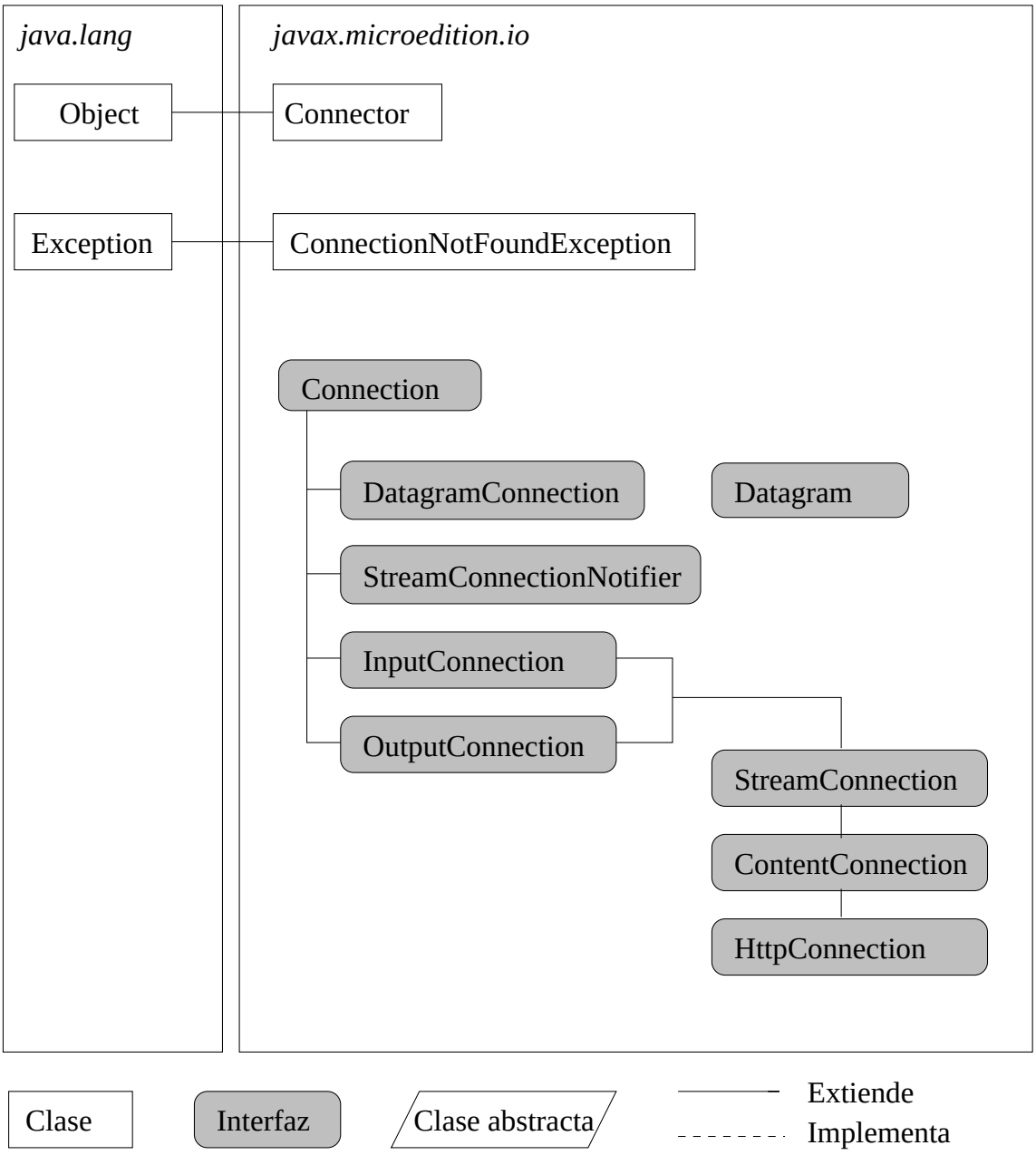
  ```
  public OutputStreamWriter(OutputStream os);
  public OutputStreamWriter(OutputStream os, String encoding)
                  throws UnsupportedEncodingException;
  ```

- Los métodos que tiene son los heredados y sobreescritos de `Writer`

11.5. Paquete javax.microedition.io

- Contiene una colección de interfaces y una clase que definen el *Generic Connection Framework*.
- Este marco es usado por los perfiles basados en CLDC, para proporcionar un mecanismo común para acceso a recursos de red y otros recursos que puedan ser direccionados con un nombre y que puedan enviar y recibir datos a través de `InputStream` y `OutputStream`.

  Un ejemplo típico de tales recursos es una página HTML o un servlet de Java, que pueden identificarse mediante un URL (Uniform Resource Locator).
- Aunque la especificación CLDC define los interfaces y métodos de este marco y sugiere cómo deben usarse para abrir conexiones a varios tipos de recursos, la especificación no requiere que sea proporcionada ninguna implementación concreta. Sin embargo, especificando los métodos comunes para abrir, cerrar y obtener datos de cualquier recurso, el marco hace un poco más fácil a los desarrolladores escribir aplicaciones que puedan conectarse a fuentes de datos usando distintos mecanismos de comunicación, tales como sockets, datagramas o HTTP, ya que sólo hay un patrón de codificación a seguir.


java.lang
javax.microedition.io
Object
Connector
Exception
ConnectionNotFoundException
Connection
DatagramConnection
Datagram
StreamConnectionNotifier
InputConnection
OutputConnection
StreamConnection
ContentConnection
HttpConnection
Clase
Interfaz
Clase abstracta
Extiende
Implementa