Tema 4. Clases Básicas Predefinidas. Entrada/Salida

Vicente Benjumea García

Programación Orientada a Objetos
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Programación Orientada a Objetos

Tema 4. Clases básicas predefinidas. Entrada/Salida

- Organización en paquetes
- Clases básicas: `java.lang`
  - La clase `java.lang.System`
  - La clase `java.lang.Math`
  - Las clases envoltorios
  - La clase `java.lang.Object`
  - Cadenas de caracteres: `String`, `StringJoiner` y `StringBuilder`
- Clases básicas del paquete `java.util`
  - La clase `java.util.Random`
  - La clase `java.util.Scanner`
- Entrada/Salida. `java.io`
  - La clase `java.nio.file.Files`
  - Lectura de fichero con `BufferedReader`
  - Lectura de fichero con `Scanner`
  - Escritura a un fichero con `PrintWriter`

Organización en paquetes

- Un **paquete** en Java es un mecanismo de **organización** que permite **clasificar** y **agrupar** clases e interfaces **relacionadas** desde un punto de vista lógico.
  - Define una jerarquía donde organizar las clases.
  - Proporciona protección de acceso.
  - Delimita un ámbito para el uso de nombres, evitan **colisiones** de nombres.
- La plataforma **Java** proporciona un conjunto de **clases, interfaces y paquetes predefinidos** para facilitar el desarrollo de clases y aplicaciones.
- Además, se pueden **definir nuevos paquetes**.
  - Cuando se crea una clase, se debe especificar el paquete al que pertenece, en la **primera sentencia** del fichero java, con la siguiente declaración:

    ```
    package <nombre-del-paquete>;
    ```

  - Si no se especifica ningún paquete, se considera que las clases e interfaces pertenecen a un **paquete anónimo**.
  - Los **ficheros**, que contienen las definiciones de las clases e interfaces, se deben situar adecuadamente en la jerarquía de carpetas, según la **jerarquía especificada por el nombre** del paquete al que pertenecen.

Uso de paquetes

- Desde **fuera** de un paquete sólo se puede acceder a clases e interfaces **públicas**.
- Para **acceder** a una clase o interfaz **desde otro paquete** se puede:
  - Utilizar el nombre de la clase **cualificado** con el nombre del paquete.

    ```
    java.util.Random rnd = new java.util.Random();
    ```

  - **Importar** la clase al comienzo del fichero y usar su nombre simple.

    ```
    import java.util.Random;
    //...
    Random rnd = new Random();
    ```

  - **Importar todas** las clases del paquete completo al comienzo del fichero y usar los nombres simples de todas las clases e interfaces del paquete.

    ```
    import java.util.*;
    //...
    Random rnd = new Random();
    ```

- El sistema de ejecución de Java **importa automáticamente** todas las clases del **paquete anónimo**, del paquete **java.lang** y del **paquete actual**.

La Interfaz de Programación de Aplicaciones de Java

- La **API** de Java (Java Application Programming Interface) es la especificación de la **biblioteca de paquetes** que se suministra con la plataforma de desarrollo de Java (**JDK**).
- Estos paquetes contienen interfaces y clases diseñadas para facilitar la tarea de programación.
- En este tema veremos parte de los paquetes: `java.lang`, `java.util` y `java.io`.

El paquete `java.lang`

- El paquete **`java.lang`** siempre está incluido en cualquier aplicación, no es necesario importarlo explícitamente.
- Contiene las clases básicas del sistema:
  - `System`: proporciona objetos para entrada/salida, y funcionalidades del sistema.
  - `Math`: proporciona funciones matemáticas usuales.
  - **Envoltorios** de tipos primitivos (`Character`, `Boolean`, `Integer`, `Double`, etc.)
  - `Object`: la **raíz** de la jerarquía de clases. Define el comportamiento base de todos los objetos.
  - `String` y `StringBuilder`: manipulación de cadenas de caracteres.
  - `Class`: proporciona información sobre las clases.
- Contiene interfaces:
  - `Comparable`: especifica el método de *ordenación natural* entre objetos (`compareTo()`).
  - `Cloneable`: especifica que un objeto puede ser *clonado*.
  - `Runnable`: especifica el método a ejecutar por el gestor de *hebras* (`run()`).
- Contiene también excepciones y errores.

La clase `java.lang.System`

- La clase `System` proporciona objetos para entrada/salida, y funcionalidades del sistema.
- Proporciona tres variables de clase (estáticas) públicas para realizar la entrada y salida de datos:
  - `InputStream in`: entrada de datos.
  - `PrintStream out`: salida de datos.
  - `PrintStream err`: salida de mensajes de error.
- Métodos de clase (estáticos) públicos:
  - `void arraycopy(...)`: permite copiar elementos de un array a otro array.
  - `long currentTimeMillis()`: devuelve el valor del tiempo actual en milisegundos.
  - `void gc()`: permite invocar al recolector de basura.
  - Etc. (consultar la documentación para más información).

La clase `java.lang.Math`

- La clase `Math` proporciona constantes numéricas y funciones matemáticas.
- Constantes (estáticas) públicas:
  - `double PI = 3.1415926535897932`: el valor del número *pi*.
  - `double E = 2.7182818284590452`: el valor del número *e*.
- Métodos de clase (estáticos) públicos:
  - `double sin(double)`, `double cos(double)`, `double tan(double)`, `double asin(double)`, `double acos(double)`, `double atan(double)`: funciones trigonométricas en radianes.
  - `x abs(x)`, `x max(x,x)`, `x min(x,x)`: valor absoluto, máximo y mínimo.
  - `double exp(double)`, `double pow(double, double)`, `double hypot(double, double)`, `double log(double)`, `double sqrt(double)`, `long round(double)`: exponenciación, potencia, hipotenusa, logaritmo, raíz cuadrada, redondeo.
  - `double random()`: generación de números aleatorios ($0 \leq r < 1$).
  - Etc. (consultar la documentación para más información).

Ejemplos de utilización de la clase `java.lang.Math`

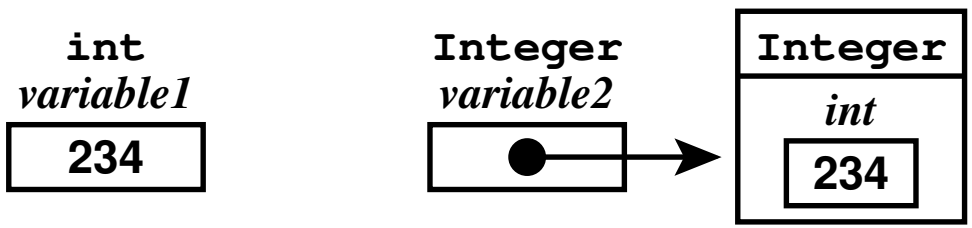
```
double x01 = Math.sin(Math.PI/2);      //  1.0
double x02 = Math.cos(Math.PI);        // -1.0
double x03 = Math.tan(Math.PI/4);      //  1.0

double x04 = Math.asin(1.0);           //  1.6
double x05 = Math.acos(-1.0);          //  3.1
double x06 = Math.atan(1.0);           //  0.8

double x07 = Math.abs(-1.0);           //  1.0     // también int
double x08 = Math.max(3.0, 4.0);       //  4.0     // también int
double x09 = Math.min(2.0, 5.0);       //  2.0     // también int

double x11 = Math.pow(Math.E, 2.0);    //  7.4
double x10 = Math.exp(2.0);            //  7.4
double x13 = Math.log(7.4);            //  2.0

double x14 = Math.sqrt(25.0);          //  5.0
double x12 = Math.hypot(3.0, 4.0);     //  5.0

long   x15 = Math.round(Math.PI);      //  3

double x16 = Math.random();            //  0 <= x16 < 1
```

Las clases envoltorios (wrappers) de `java.lang`

- Los valores de los tipos primitivos no son objetos.
  - Una variable de tipo primitivo **almacena el valor** en sí.
  - Una variable de tipo objeto **almacena una referencia** al objeto.
- A veces es útil manipular valores de tipos primitivos como si fueran objetos.
- Para cada **tipo primitivo** existe una **clase envoltorio** en el paquete `java.lang`.

`char`	`boolean`	byte	short	`int`	long	float	`double`
`Character`	`Boolean`	Byte	Short	`Integer`	Long	Float	`Double`

- Las clases envoltorios permiten manipular valores de tipos primitivos como objetos.
- Los objetos de las clases envoltorios son **inmutables**.
- Se **envuelve** y **desenvuelve automáticamente** cuando sea necesario (asignaciones, castings y paso de parámetros).



Metodos de clase de los envoltorios

- **Ayudan en la manipulación de valores de tipos primitivos.**
- El método de clase `parseXxx()` crea valores primitivos a partir de `String`.
  - `boolean b = Boolean.parseBoolean("true");`
  - `int i = Integer.parseInt("234");` `// lanza NumberFormatException`
  - `double d = Double.parseDouble("34.67");` `// lanza NumberFormatException`
- El método de clase `toString()` crea `String` a partir de valores primitivos.
  - `String c = Character.toString('a');`
  - `String b = Boolean.toString(true);`
  - `String i = Integer.toString(234);`
  - `String d = Double.toString(34.67);`
- El método de clase `hashCode()` devuelve el *código hash* de valores primitivos.
  - `int hc = Character.hashCode('a');`
  - `int hb = Boolean.hashCode(true);`
  - `int hi = Integer.hashCode(234);`
  - `int hd = Double.hashCode(34.67);`
- El método de clase `compare()` compara valores primitivos y devuelve $<$ $==$ $>$ que 0.
  - `int cc = Character.compare('a', 'b');` `// < 0`
  - `int cb = Boolean.compare(true, false);` `// > 0`
  - `int ci = Integer.compare(234, 456);` `// < 0`
  - `int od = Double.compare(34.67, 12.7);` `// > 0`
- Metodos de clase adicionales del envoltorio `Character`.
  - `boolean b = Character.isDigit(char);`
  - `boolean b = Character.isLetter(char);`
  - `boolean b = Character.isLowerCase(char);`
  - `boolean b = Character.isUpperCase(char);`
  - `boolean b = Character.isSpaceChar(char);`
  - `char c = Character.toLowerCase(char);`
  - `char c = Character.toUpperCase(char);`

Metodos de instancia de los envoltorios

- Creación y manipulación de objetos envoltorios (envuelven valores de tipos primitivos).
- Java **envuelve** y **desenvuelve automáticamente** cuando sea necesario.

```
Character oc = 'a';   | (Character) 'a'   || char    c = oc;  | (char) oc
Boolean   ob = false; | (Boolean) false   || boolean b = ob;  | (boolean) ob
Integer   oi = 234;   | (Integer) 234     || int     i = oi;  | (int) oi
Double    od = 34.67; | (Double) 34.67    || double  d = od;  | (double) od
```

- El método de instancia `toString()` crea `String` a partir del valor del objeto.

```
String c = oc.toString();
String b = ob.toString();
String i = oi.toString();
String d = od.toString();
```

- El método de instancia `hashCode()` devuelve el *código hash* del objeto.

```
int hc = oc.hashCode();
int hb = ob.hashCode();
int hi = oi.hashCode();
int hd = od.hashCode();
```

- El método de instancia `compareTo()` compara objetos y devuelve $<$ $==$ $>$ que cero.

```
int cc = oc.compareTo(oc);      | int cc = oc.compareTo('a');
int cb = ob.compareTo(ob);      | int cb = ob.compareTo(false);
int ci = oi.compareTo(oi);      | int ci = oi.compareTo(234);
int cd = od.compareTo(od);      | int cd = od.compareTo(34.67);
```

- El método de clase `valueOf()` crea objetos envoltorios a partir de valores o `String`.

```
Character oc = Character.valueOf('a');   | oc = Character.valueOf('a'); | oc = 'a';
Boolean   ob = Boolean.valueOf("false"); | ob = Boolean.valueOf(false); | ob = false;
Integer   oi = Integer.valueOf("234");   | oi = Integer.valueOf(234);   | oi = 234;
Double    od = Double.valueOf("34.67");  | od = Double.valueOf(34.67);  | od = 34.67;
```

Envolver y desenvolver automáticamente

- Cuando sea necesario, el compilador realiza de forma **automática** la **conversión** de valores de tipos primitivos a objetos envoltorios y viceversa (**no para == o !=**).

```
Double[] lista = new Double[20];
lista[i] = 45.5;                    // Envuelve automáticamente
double d = 5.2 + lista[j];          // Desenvuelve automáticamente

Integer v1 = Integer.parseInt("123");
Integer v2 = 127;
v1 += 5;
++v2;
System.out.println(v1+" < "+v2+" "+(v1<v2));     // 128 < 128  false
System.out.println(v1+" >= "+v2+" "+(v1>=v2));   // 128 >= 128 true
System.out.println(v1+" == "+v2+" "+(v1==v2));   // 128 == 128 false ERROR
```

- Tanto el método de instancia `compareTo(T)` como el método de clase `compare(T,T)` permiten comparar dos objetos o valores primitivos. Devuelven `< == >` que cero.
- La comparación de igualdad se debe hacer con `equals(o)`, los operadores `==` y `!=` comparan las referencias a los objetos, no los valores.
- La asignación (=) de variables de tipos *envoltorios* **asigna las referencias** a los objetos (compartidos).
- La asignación (=) de variables de tipos *primitivos* **asigna los valores**.

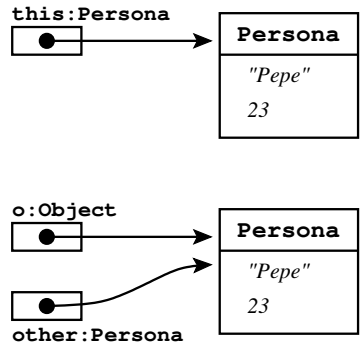
La clase `java.lang.Object`

- Es la clase **raíz** de toda la jerarquía de clases de Java.
  - Todas las clases heredan, directa o indirectamente, de `Object`.
  - Si una clase no hereda de otra, entonces **hereda automaticamente** de `Object`.
  - La clase `Object` define el comportamiento mínimo **común** de todos los objetos.
  - No tiene mucho sentido crear instancias de `Object`, ya que la funcionalidad que ofrece es limitada.
- Métodos de instancia importantes de la clase `Object`:
  - `String toString()`: representación `String` del objeto actual.
  - `boolean equals(Object o)`: indica si otro objeto es *igual* al objeto actual.
  - `int hashCode()`: devuelve el valor del código hash del objeto actual.
  - `Object clone()`: crea un nuevo objeto y devuelve una copia (bit a bit) del objeto actual.
  - `Class getClass()`: devuelve información sobre la clase del objeto actual.
  - Etc. (consultar la documentación para más información).
- Usualmente **NO es adecuada** la implementación por defecto, y en caso de ser necesario un determinado método, usualmente las subclases deben redefinir su implementación.

La clase java.lang.Object. El método equals()

- `boolean equals(Object o)`: indica si el **contenido** de otro objeto es **igual** al **contenido** del objeto actual.
- Usualmente, las clases derivadas deben **redefinir** este método para que realice la comparación adecuadamente (por **defecto** realiza una comparación por ==).
- Todas las **clases del sistema** tienen redefinido este método (también `enum`).
- La anotación `@Override` indica al compilador que un método pretende redefinir a un método de la superclase (útil para detectar errores).
- El operador `instanceof` comprueba si un objeto es una **instancia de** una clase.

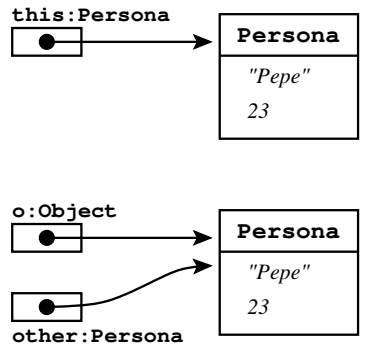
```
class Persona {
    private String nombre;
    private int edad;
    public Persona(String n, int e) { nombre = n; edad = e; }
    @Override
    public boolean equals(Object o) {
        boolean ok = false;
        if (o instanceof Persona) {
            Persona other = (Persona) o;
            ok = (this.edad == other.edad) && this.nombre.equals(other.nombre);
        }
        return ok;  // compara DIFERENCIANDO mayúsculas y minúsculas.
    }
}
```



La clase java.lang.Object. El método equals()

- `boolean equals(Object o)`: indica si el **contenido** de otro objeto es **igual** al **contenido** del objeto actual.
- Usualmente, las clases derivadas deben **redefinir** este método para que realice la comparación adecuadamente (por **defecto** realiza una comparación por ==).
- Todas las **clases del sistema** tienen redefinido este método (también `enum`).
- La anotación `@Override` indica al compilador que un método pretende redefinir a un método de la superclase (útil para detectar errores).
- El operador `instanceof` comprueba si un objeto es una **instancia de** una clase.

```
class Persona {
    private String nombre;
    private int edad;
    public Persona(String n, int e) { nombre = n; edad = e; }
    @Override
    public boolean equals(Object o) {
        boolean ok = false;
        if (o instanceof Persona) {
            Persona other = (Persona) o;
            ok = (this.edad == other.edad) && this.nombre.equalsIgnoreCase(other.nombre);
        }
        return ok;  // compara SIN DIFERENCIAR mayúsculas y minúsculas.
    }
}
```



La clase java.lang.Object. El método hashCode()

- Si dos objetos son **iguales** (según el método `equals()`), entonces el método `hashCode()` debe devolver el **mismo** valor del **código hash** para ambos objetos. Si `a.equals(b)` entonces `a.hashCode() == b.hashCode()`.
  - Es posible que dos objetos diferentes generen el mismo código hash. Se denomina **colisión**.
- Si se redefine el método `equals()`, entonces también se debe redefinir el método `hashCode()`, considerando los mismos componentes (por **defecto** se calcula a partir de la referencia del objeto).
- `int hashCode()`: devuelve el valor del **código hash** del objeto actual. Se calcula como la **suma** de los `hashCode()` de los componentes involucrados en `equals()`.

```
class Persona {
    private String nombre;
    private int edad;
    public Persona(String n, int e) { nombre = n;  edad = e; }
    @Override
    public boolean equals(Object o) {
        boolean ok = false;
        if (o instanceof Persona) {
            Persona other = (Persona) o;
            ok = (this.edad == other.edad) && this.nombre.equals(other.nombre);
        }
        return ok;  // compara DIFERENCIANDO mayúsculas y minúsculas.
    }
    @Override
    public int hashCode( ) {
        return java.util.Objects.hash(this.edad, this.nombre);
        //  return Integer.hashCode(this.edad) + this.nombre.hashCode();
    }
}
```

La clase java.lang.Object. El método hashCode()

- Si dos objetos son **iguales** (según el método `equals()`), entonces el método `hashCode()` debe devolver el **mismo** valor del **código hash** para ambos objetos. Si `a.equals(b)` entonces `a.hashCode() == b.hashCode()`.
  - Es posible que dos objetos diferentes generen el mismo código hash. Se denomina **colisión**.
- Si se redefine el método `equals()`, entonces también se debe redefinir el método `hashCode()`, considerando los mismos componentes (por **defecto** se calcula a partir de la referencia del objeto).
- `int hashCode()`: devuelve el valor del **código hash** del objeto actual. Se calcula como la **suma** de los `hashCode()` de los componentes involucrados en `equals()`.

```
class Persona {
    private String nombre;
    private int edad;
    public Persona(String n, int e) { nombre = n;  edad = e; }
    @Override
    public boolean equals(Object o) {
        boolean ok = false;
        if (o instanceof Persona) {
            Persona other = (Persona) o;
            ok = (this.edad == other.edad) && this.nombre.equalsIgnoreCase(other.nombre);
        }
        return ok;  // compara SIN DIFERENCIAR mayúsculas y minúsculas.
    }
    @Override
    public int hashCode() {
        return java.util.Objects.hash(this.edad, this.nombre.toLowerCase());
        //  return Integer.hashCode(this.edad) + this.nombre.toLowerCase().hashCode();
    }
}
```

Cadenas de caracteres

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

```
"Ejemplo de cadena de caracteres"
```

Caracteres especiales:

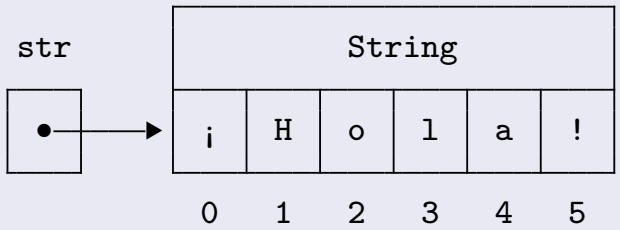
- El **salto de línea** (o **nueva línea**) se representa mediante el carácter (`char`) `'\n'`, y mediante la cadena de caracteres (`String`) `"\n"`. Ej.: `"Hola\n"`.
- El **tabulador** se representa mediante el carácter (`char`) `'\t'`, y mediante la cadena de caracteres (`String`) `"\t"`. Ej.: `"\tHola"`.
- Otros caracteres especiales: **retorno de carro** (`'\r'`), **nueva página** (`'\f'`), **barra invertida** (`'\\'`), **comilla simple** (`'\''`), **comilla doble** (`'\"'`).

Para manipular cadenas de caracteres, por razones de eficiencia, se utilizan dos clases incluidas en `java.lang` y una de `java.util`:

- `java.lang.String`: para representar cadenas inmutables
- `java.util.StringJoiner`: para unir cadenas con separadores.
- `java.lang.StringBuilder`: para manipular cadenas modificables.

La clase java.lang.String

- Cada objeto de la clase `String` contiene una cadena de caracteres.
- Los objetos de esta clase se pueden inicializar:
  - Forma normal:
    ```
    String str = new String("¡Hola!");
    ```
  - Forma simplificada:
    ```
    String str = "¡Hola!";
    ```



- Los objetos de la clase `String` en Java son **inmutables**.
  - Una cadena de caracteres almacenada en un objeto de la clase `String` **NO** puede ser modificada (crecer, cambiar un carácter, etc.).
- Sin embargo, a una variable de la clase `String` se le pueden asignar objetos `String` distintos durante su existencia.
- Si se intenta acceder a una posición no válida del String, el sistema lanza la excepción `StringIndexOutOfBoundsException`.

La clase java.lang.String

Métodos de Clase para Conversiones a/desde String

- `String String.valueOf(Tipo v)`: devuelve un nuevo objeto `String` con la representación del valor especificado en `v` (de tipo `boolean`, `char`, `int`, `double`).
- `String Integer.toString(int v)`: devuelve un nuevo objeto `String` con la representación del valor especificado en `v` (de tipo `int`).
- `String Double.toString(double v)`: devuelve un nuevo objeto `String` con la representación del valor especificado en `v` (de tipo `double`).
- `int Integer.parseInt(String s)`: devuelve el resultado de obtener el valor del número entero representado en `s` (de tipo `String`).
- `double Double.parseDouble(String s)`: devuelve el resultado de obtener el valor del número real representado en `s` (de tipo `String`).

```
String b = String.valueOf('a');          String e = Character.toString('a');   // "a"
String a = String.valueOf(true);         String f = Boolean.toString(true);    // "true"
String c = String.valueOf(123);          String g = Integer.toString(123);     // "123"
String d = String.valueOf(123.456);      String h = Double.toString(123.456);  // "123.456"

boolean i = Boolean.parseBoolean("true");   // i toma el valor true
int     j = Integer.parseInt("123");        // j toma el valor 123     Error si " 123 "
double  k = Double.parseDouble("123.456");  // k toma el valor 123.456
```

La clase java.lang.String

Metodos de consulta

- `int length()`: devuelve el número de caracteres de la cadena.
- `char charAt(int p)`: devuelve el char en la posición p de la cadena ($0 \leq p < length()$).
- `int indexOf(char/String c2)`: devuelve la primera posición de c2.
- `int lastIndexOf(char/String c2)`: devuelve la última posición de c2.
- `int indexOf(char/String c2, int d)`: devuelve prim. pos. de c2 desde la pos. d.
- `int lastIndexOf(char/String c2, int d)`: devuelve últ. pos. de c2 desde la pos. d.
- `boolean contains(String c2)`: devuelve `true` si c2 se encuentra en la cadena.

```
String  str = "programas";
char    a = str.charAt(0);                // a toma el valor 'p'
char    b = str.charAt(str.length()-1); // b toma el valor 's'

int     c = str.length();                 // c toma el valor  9
int     d = str.indexOf("xxx");           // d toma el valor -1
int     e = str.indexOf('r');             // e toma el valor  1
int     f = str.indexOf('r', 2);          // f toma el valor  4
int     g = str.lastIndexOf('r');         // g toma el valor  4
int     h = str.lastIndexOf('r', 3);      // h toma el valor  1
boolean x = str.contains("gram");         // x toma el valor  true
```

La clase java.lang.String

Metodos que producen nuevos objetos String

- `String substring(int desde, int hasta)`: devuelve un nuevo objeto `String` con los caracteres a partir de la posición `desde` y hasta la posición `hasta` (sin incluirla).
- `String substring(int desde)`: devuelve un nuevo objeto `String` con los caracteres a partir de la posición `desde` y hasta el final.
- `String replace(String c2, String c3)`: devuelve un nuevo objeto `String` con los caracteres del objeto actual, reemplazando todas las ocurrencias de c2 por c3.

```
String str = "tuvo un coche que mantuvo";
String a = str.substring(8, 13);          // a toma el valor "coche"
String b = str.substring(14);             // b toma el valor "que mantuvo"

String c = str.replace("tuvo", "tiene"); // c toma el valor "tiene un coche que mantiene"
```

La clase java.lang.String

Metodos que producen nuevos objetos String

- `String toLowerCase()`: devuelve un nuevo objeto `String` con todos los caracteres del objeto actual en minúsculas.
- `String toUpperCase()`: devuelve un nuevo objeto `String` con todos los caracteres del objeto actual en mayúsculas.
- `String trim()`: devuelve un nuevo objeto `String` con todos los caracteres del objeto actual, pero elimina los espacios iniciales y finales.
- `String concat(String c2)`: devuelve un nuevo objeto `String` con los caracteres del objeto actual concatenados con los caracteres de c2 (equiv a +).
- La concatenación (+ y +=) produce un nuevo objeto `String` como resultado de unir dos cadenas de caracteres.

```
String str = "▫▫ProGraMas▫▫";          // el símbolo ▫ representa el espacio
String a = str.toLowerCase();          // a toma el valor "▫▫programas▫▫"
String b = str.toUpperCase();          // b toma el valor "▫▫PROGRAMAS▫▫"

String c = str.trim();                 // c toma el valor "ProGraMas"

String d = str.concat("Dobles");       // d toma el valor "▫▫ProGraMas▫▫Dobles"
String e = str + "Dobles";             // e toma el valor "▫▫ProGraMas▫▫Dobles"
```

La clase java.lang.String

Código Hash y Metodos de comparación

- `int hashCode()`: devuelve el valor del código hash del objeto actual.
- `boolean equals(String c2)`: devuelve `true` si el objeto actual y c2 son iguales y `false` en otro caso.
- `boolean equalsIgnoreCase(String c2)`: igual que la anterior, pero sin tener en cuenta las diferencias por mayúsculas y minúsculas.
- `int compareTo(String c2)`: devuelve un entero *menor*, *igual* o *mayor* que **cero** cuando el objeto actual es *menor*, *igual* o *mayor* que c2.
- `int compareToIgnoreCase(String c2)`: igual que la anterior, pero sin tener en cuenta las diferencias por mayúsculas y minúsculas.
- El operador (==) compara si ambas variables referencian al mismo objeto.

```
String str1 = "programas";
String str2 = "PROGRAMAS";
int a = str1.hashCode();                    // -968779338
int b = str2.hashCode();                    //  1817815446
int c = str2.toLowerCase().hashCode();      // -968779338

boolean d = str1.equals(str2);              // false
boolean e = str1.equalsIgnoreCase(str2);    // true

int f = str1.compareTo(str2);               //  > 0
int g = str1.compareToIgnoreCase(str2);     // == 0
```

La clase java.lang.String

El metodo split()

- `String[] split(String regex)`: devuelve un array de `String` resultado de dividir la cadena de caracteres del objeto actual en substrings según los delimitadores especificados por `regex`.
- `String[] split(String regex, int limit)`: igual que el anterior, pero con un límite en la aplicación de la división.
- Los delimitadores son expresiones regulares. Por ejemplo:
  - `"[,;:]"` // *Exactamente uno de entre ,;:*
  - `"[,;:]+"` // *Uno o más de entre ,;:*
  - `"[^a-zA-Z0-9]"` // *Cualquier carácter que no sea una letra o dígito*
  - `"[,;:] ?"` // *Uno de entre ,;: seguido por un espacio opcional*
  - `"\\s*[,;:]\\s*"` // *Uno de entre ,;: entre múltiples [\t\n\r\f] opcionales*

```
String   s1 = "33456765M Juan 6.5";
String[] t1 = s1.split("[ ]");              // [ "33456765M", "Juan", "6.5" ]

String   s2 = "El   agua::es;buena";
String[] t2 = s2.split("[ :]");             // [ "El", "", "", "agua", "", "es;buena" ]
String[] t3 = s2.split("[ :]+");            // [ "El", "agua", "es;buena" ]
String[] t4 = s2.split("[ :]+", 2);         // [ "El", "agua::es;buena" ]

String   s5 = "El agua:es ;   buena";
String[] t5 = s5.split("\\s*[,;:]\\s*");    // [ "El agua", "es", "buena" ]
```

La clase java.lang.String

El metodo split(). Ejemplo

```
String numeros = " 1.23 ; 2.34 ; 3,45 ; xxx ; 5.67 ";
String[] datos = numeros.split("\\s*[;]\\s*");  // [ "1.23", "2.34", "3,45", "xxx", "5.67
for (String x : datos) {                        // Muestra en consola:
    try {                                       // 1.23
        double valor = Double.parseDouble(x);   // 2.34
        System.out.println(valor);              // Error 3,45
    } catch (NumberFormatException e) {         // Error xxx
        System.out.println("Error " + x);       // 5.67
    }
}
```

```
String linea = " pepe : 1.23 ; ana : 2.34 ; juan : 3,45 ; xxx ; eva : 5.67 ";
String[] datos = linea.split("\\s*[;]\\s*");
for (String persona : datos) {
    try {
        String[] p = persona.split("\\s*[:]\\s*");      // Muestra en consola:
        String nombre = p[0];                           // pepe -> 1.23
        double valor = Double.parseDouble(p[1]);        // ana -> 2.34
        System.out.println(nombre + " -> " + valor);    // Error juan : 3,45
    } catch (Exception e) {                             // Error xxx
        System.out.println("Error " + persona);         // eva -> 5.67
    }
}
```

La clase java.lang.String

El método de clase (estático) join()

- Crea un nuevo objeto `String` como resultado de **unir** al resto de parámetros (de tipo `String`, o array de `String`, o lista de `String`), utilizando al primer parámetro como **separador** (de tipo `String`).

```
String[] datos = { "Pepe", "Lola", "María" };
String s1 = String.join("; ", datos);                    // "Pepe; Lola; María"

String s2 = String.join("; ", "Pepe", "Lola", "María"); // "Pepe; Lola; María"

String s3 = String.join("; ", "Pepe", null, "María");   // "Pepe; null; María"


List<String> datos2 = new ArrayList<>(Arrays.asList(datos));
String s4 = String.join("; ", datos2);                    // "Pepe; Lola; María"
```

La clase java.lang.String

El método de clase (estático) format()

- Crea un nuevo objeto `String` como resultado de aplicar el **descriptor del formato** al resto de parámetros.
- Los formatos se especifican con el *carácter de escape* **%**, seguido opcionalmente por una especificación de *anchura de campo* y una *precisión*, y finalmente una descripción del tipo de datos del valor:
  - %b: valores lógicos (`boolean`).
  - %c: caracteres (`char`).
  - %d: números enteros (`byte`, `short`, `int` y `long`).
  - %f: números reales en punto fijo (`float`, `double`).
  - %e: números reales en notación científica (`float`, `double`).
  - %g: números reales en punto flotante (`float`, `double`).
  - %s: cualquier objeto. Utiliza `toString()` o `String.valueOf()`.

```
String.format("texto [%b] [%c]", true, 'a')                  // texto [true] [a]
String.format("[%s] [%6s] [%-6s]", "hola", "hola", "hola")   // [hola] [  hola] [hola  ]
String.format("[%d] [%6d] [%06d]", 2345, 2345, 2345)         // [2345] [  2345] [002345]
String.format("[%f] [%9.2f]", 12.356, 12.356)                // [12,356000] [    12,36]
String.format("[%e] [%9.2e]", 12.356, 12.356)                // [1,235600e+01] [ 1,24e+01]
String.format("[%g] [%9.2g]", 12.356, 12.356)                // [12,3560] [       12]
String.format("[%g] [%9.2g]", 0.00009, 123.56)               // [9,00000e-05] [  1,2e+02]
String.format(Locale.US, "[%.2f] [%.2e] [%.2g]", 1.0, 1.0, 1.0) // [1.00] [1.00e+00] [1.0]
```

La clase java.lang.String

El método printf() de PrintStream y PrintWriter

- El método `printf(String formato, ...)` de las clases `PrintStream` y `PrintWriter` muestra una cadena de caracteres con formato.

```
class A {
    private int a;
    public A(int s){ a = s; }
    public String toString() { return "A(" + a + ")"; }
}

public static void main(String[] args) {
    A obj = new A(65);
    System.out.printf("texto [%b] [%c] [%s]\n", true, 'a', obj);
    System.out.printf("[%s] [%6s] [%-6s]\n", "hola", "hola", "hola");
    System.out.printf("[%d] [%6d] [%06d]\n", 2345, 2345, 2345);
    System.out.printf(Locale.FRANCE, "[%.2f] [%.2e] [%.2g]\n", 1.0, 1.0, 1.0);
    System.out.printf(Locale.UK, "[%.2f] [%.2e] [%.2g]\n", 1.0, 1.0, 1.0);
}
```

```
texto [true] [a] [A(65)]
[hola] [  hola] [hola  ]
[2345] [  2345] [002345]
[1,00] [1,00e+00] [1,0]
[1.00] [1.00e+00] [1.0]
```

La clase java.util.StringJoiner

- La clase `StringJoiner` resulta útil para crear una nueva cadena de caracteres como resultado de **unir** varias cadenas de caracteres, de tal forma que la cadena resultante se crea utilizando un *prefijo*, un *sufijo*, y un *separador* que permite separar las cadenas que han sido unidas.
- El constructor `StringJoiner(String, String, String)` permite especificar tanto el *separador*, como el *prefijo* y el *sufijo* (en ese orden). El *prefijo* y el *sufijo* se pueden omitir: `StringJoiner(String)`
- El método `add(String)` permite añadir la cadena de caracteres especificada.
- En caso de que el objeto a añadir (add) no sea de tipo `String`, entonces se deberá utilizar su representación textual `String.valueOf(x)` o `x.toString()`.
- El método `toString()` devuelve el resultado de unir todas las cadenas especificadas, considerando el separador, prefijo y sufijo especificados.

```
String[] datos = { "Pepe", "Lola", "María" };
StringJoiner sj = new StringJoiner("; ", "[ ", " ]");
for (String x : datos) {
    sj.add(x);          // sj.add(x.toString()); // sj.add(String.valueOf(x));
}
String res = sj.toString(); // "[ Pepe; Lola; María ]"
```

La clase java.lang.StringBuilder

- La clase `StringBuilder` permite **manipular** cadenas de caracteres.
  - Cada objeto de la clase `StringBuilder` contiene una cadena de caracteres.
  - Las cadenas de caracteres de los objetos `StringBuilder` se pueden ampliar, reducir y modificar mediante métodos.
  - Cuando la capacidad establecida se excede, se aumenta automáticamente.
- Los objetos de esta clase se construyen de las siguientes formas:

```
StringBuilder sb1 = new StringBuilder();        // capacidad por defecto
StringBuilder sb2 = new StringBuilder(10);      // capacidad inicial
StringBuilder sb3 = new StringBuilder("hola"); // valor inicial
```

- Métodos de consulta:

```
int  length();                      // número de caracteres almacenados
char charAt(int pos);               // carácter de la posición especificada
int  indexOf(String str);           // posición de la primera ocurrencia
int  indexOf(String str, int d);    // posición de la primera ocurrencia desde
int  lastIndexOf(String str);       // posición de la última ocurrencia
int  lastIndexOf(String str, int d); // posición de la última ocurrencia desde
```

- Métodos para construir objetos String:

```
String substring(int desde, int hasta);  // substring desde sin incluir hasta
String substring(int desde);             // substring desde hasta el final
String toString();                       // representación textual
```

La clase java.lang.StringBuilder

- Métodos para modificar objetos StringBuilder:
  - `void setLength(int nl);` // cambia la longitud de la cadena
  - `void setCharAt(int pos, char car);` // modificar carácter de posición
  - `StringBuilder append(String str);` // añadir string al final
  - `StringBuilder insert(int pos, String str);` // insertar string en posición
  - `StringBuilder deleteCharAt(int pos);` // eliminar carácter de posición
  - `StringBuilder delete(int desde, int hasta);` // eliminar desde sin incl. hasta
  - `StringBuilder replace(int desde, int hasta, String str);` // reemplazar
  - `StringBuilder reverse();` // invertir el string almacenado

```
String[] datos = { "Pepe", "Lola", "María" };
StringBuilder sb = new StringBuilder();
sb.append("[ ");
if (datos.length > 0) {
    sb.append(datos[0]);
    for (int i = 1; i < datos.length; ++i) {
        sb.append("; ");
        sb.append(datos[i]);
    }
}
sb.append(" ]");
String res = sb.toString(); // "[ Pepe; Lola; María ]"
```

La clase java.lang.StringBuilder

- Ejemplo:

```
public class StringBuilderDemo {
    public static void main(String[] args) {
        final String FRASE = "TUVO un coche que mantuvo";
        final String ANTER = "tuvo";
        final String NUEVA = "tiene";
        StringBuilder sb = new StringBuilder(FRASE);
        int i = 0;
        while (i+ANTER.length() <= sb.length()) {
            if (ANTER.equalsIgnoreCase(sb.substring(i, i+ANTER.length()))) {
                sb.replace(i, i+ANTER.length(), NUEVA);
                i += NUEVA.length();
            } else {
                ++i;
            }
        }
        for (int i = 0; i < sb.length(); i++) {
            if (Character.isLowerCase(sb.charAt(i))
                && ((i == 0) || ! Character.isLetter(sb.charAt(i-1)))) {
                sb.setCharAt(i, Character.toUpperCase(sb.charAt(i)));
            }
        }
        sb.append(" en el garage.");
        System.out.println(sb);        // Tiene Un Coche Que Mantiene en el garage.
    }
}
```

El paquete `java.util`

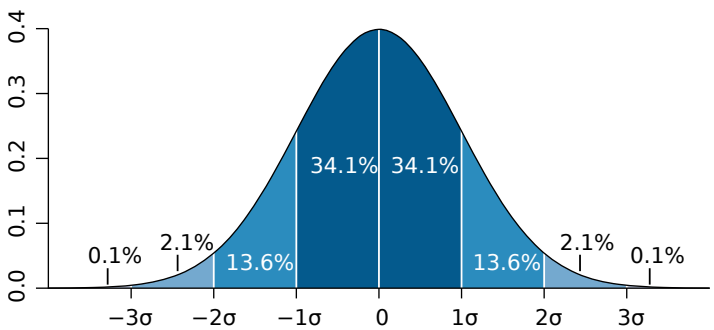
- El paquete `java.util` contiene clases de utilidades.
  - La clase `Random`: generador de números aleatorios.
  - La clase `Scanner`: entrada de datos.
  - Las colecciones: colecciones de datos organizadas adecuadamente (tema 6).
  - Contiene también interfaces y excepciones.
  - Etc. (consultar la documentación para más información).

La clase java.util.Random

- La clase `Random` proporciona un generador de números pseudo-aleatorios.
- Los objetos de la clase random permiten generar números pseudo-aleatorios que siguen una determinada distribución de probabilidad.

```
java.util.Random rnd = new java.util.Random();
```

- Los siguientes metodos de instancia permiten generar números aleatorios que siguen un *distribución de probabilidad uniforme*.
  - `int nextInt(int n);` `// 0 <= r < n`
  - `int nextInt();` `// Integer.MIN_VALUE <= r <= Integer.MAX_VALUE`
  - `long nextLong();` `// Long.MIN_VALUE <= r <= Long.MAX_VALUE`
  - `float nextFloat();` `// 0.0 <= r < 1.0`
  - `double nextDouble();` `// 0.0 <= r < 1.0 equiv. double Math.random()`
  - `boolean nextBoolean();` `// false, true`
- El siguiente metodo de instancia permite generar números aleatorios que siguen un *distribución de probabilidad normal* (gausiana) con media 0 y desviación típica 1.
  - `double nextGaussian();` `// media 0 y desviación típica 1 (-3 < 99.7% < 3)`



Ejemplo de uso de la clase java.util.Random

Ejemplo de generación de números aleatorios enteros entre 0 y 9 (ambos inclusive):

```java
public class EjemploRandom1 {

  private int[] datos;
  public EjemploRandom1(int nelms) {
    datos = new int[nelms];
    for (int i = 0; i < datos.length; ++i) {
      datos[i] = (int)(10 * Math.random());
    }
  }
}
```

```java
import java.util.Random;
public class EjemploRandom2 {
private static Random rnd = new Random();
private int[] datos;
  public EjemploRandom2(int nelms) {
    datos = new int[nelms];
    for (int i = 0; i < datos.length; ++i) {
      datos[i] = (int)(10 * rnd.nextDouble());
    }
  }
}
```

```java
import java.util.Random;
public class EjemploRandom3 {
  private static Random rnd = new Random();
  private int[] datos;
  public EjemploRandom3(int nelms) {
    datos = new int[nelms];
    for (int i = 0; i < datos.length; ++i) {
      datos[i] = rnd.nextInt(10);
    }
  }
}
```

```java
// Gen. dist. Normal números reales entre 0 y 10
import java.util.Random;
public class EjemploRandom4 {
  private static Random rnd = new Random();
  private double[] datos;
  public EjemploRandom4(int nelms) {
    datos = new double[nelms];
    for (int i = 0; i < datos.length; ++i) {
      datos[i] = rnd.nextGaussian()*5.0/3.0+5.0;
      if (datos[i] < 0.0) { datos[i] = 0.0; }
      if (datos[i] > 10.0) { datos[i] = 10.0; }
    }
  }
}
```

La clase java.util.Scanner

- La clase `Scanner` permite crear objetos asociados a flujos de entrada para realizar una entrada de datos fácil y flexible.

```
Scanner sc1 = new Scanner("Datos desde String");          // datos desde String
Scanner sc2 = new Scanner(System.in);                     // datos desde teclado
Scanner sc3 = new Scanner(Path.of("nombre_fichero.txt")); // datos desde fichero
```

- Se debe **cerrar** el *objeto scanner* cuando finaliza la entrada de texto. `void close();`
- La sentencia **"try-con-recursos"** permite **cerrar automáticamente** el objeto *scanner* al finalizar la entrada de texto, tanto si la ejecución termina con éxito, como si se produce alguna excepción.

```
import java.util.Scanner;
class EjScanner {
    public static void main(String[] args) {
        try (Scanner sc = new Scanner("Pepe 20 María 30")) {
            // ...
        } catch (InputMismatchException e) {
            // ...
        } catch (NoSuchElementException e) {
            // ...
        }
    }
}
```

La clase java.util.Scanner

- Proporciona los siguientes métodos para la entrada de **tokens**:
  - `String  next();        // lee y devuelve el siguiente token como String`
  - `int     nextInt();     // lee y devuelve el siguiente token como int`
  - `double  nextDouble();  // lee y devuelve el siguiente token como double`
  - `boolean nextBoolean(); // lee y devuelve el siguiente token como boolean`
    - Los delimitadores para **tokens** son espacios, tabs y nueva línea (con repetición) `"[ \t\n\r\f]+"`
- Proporciona el siguiente método para la entrada de **líneas**:
  - `String  nextLine();    // lee y devuelve la siguiente línea como String`
    - El delimitador para **líneas** es la nueva línea (sin repetición) `"\r\n|[\n\r\u2028\u2029\u0085]"`
- Se recomienda **NO** utilizar el mismo objeto `Scanner` para entrada de **tokens** y de **líneas** simultáneamente.
- Lanzan las siguientes excepciones en caso de error:
  - `NoSuchElementException     // si no hay más datos disponibles`
  - `InputMismatchException     // si el dato a leer no es del tipo esperado`

```
try (Scanner sc = new Scanner("Pepe 20 María 30")) {
    String nombre1 = sc.next();
    int    edad1   = sc.nextInt();
    String nombre2 = sc.next();
    int    edad2   = sc.nextInt();
    System.out.println(nombre1 + ": " + edad1);     // Pepe: 20
    System.out.println(nombre2 + ": " + edad2);     // María: 30
} catch (InputMismatchException e) {
    System.out.println("Error al extraer la edad");
} catch (NoSuchElementException e) {
    System.out.println("Error al extraer el dato");
}
```

La clase java.util.Scanner

- Proporciona los siguientes métodos de instancia para comprobar si hay datos disponibles del tipo adecuado:
  - `boolean hasNext();` // devuelve true hay un token disponible
  - `boolean hasNextInt();` // devuelve true si el siguiente token es int
  - `boolean hasNextDouble();` // devuelve true si el siguiente token es double
  - `boolean hasNextBoolean();` // devuelve true si el siguiente token es boolean
  - `boolean hasNextLine();` // devuelve true hay una línea disponible

- Se recomienda **NO** utilizar el mismo objeto `Scanner` para entrada de **tokens** y de **líneas** simultáneamente.

```
try (Scanner sc = new Scanner("Pepe 20 María 30 Juan 25 Lola 22")) {
    String nombre;
    int edad;
    while (sc.hasNext()) {
        nombre = sc.next();
        if (sc.hasNextInt()) {
            edad = sc.nextInt();
        } else {                                        // Pepe: 20
            edad = 0;                                   // María: 30
        }                                               // Juan: 25
        System.out.println(nombre + ": " + edad);       // Lola: 22
    }
}
```

La clase java.util.Scanner

- Ejemplo de entrada de líneas:

```
String lineas = "Esta es la primera línea\n"
              + "Esta es la segunda línea\n"
              + "Esta es la última línea";
try (Scanner sc = new Scanner(lineas)) {
    while (sc.hasNextLine()) {                    // [Esta es la primera línea]
        String linea = sc.nextLine();             // [Esta es la segunda línea]
        System.out.println("[" + linea + "]");   // [Esta es la última línea]
    }
}
```

La clase java.util.Scanner

- Se puede especificar la localización para la lectura de números reales.
  - La localización española y francesa utilizan la **coma decimal**.
  - La localización británica utiliza el **punto decimal**.
  - `Scanner useLocale(Locale locale); // Locale.ENGLISH, Locale.FRENCH`

```
try (Scanner sc = new Scanner("Pepe 7.2 María 8.5")) {
    sc.useLocale(Locale.ENGLISH);   // punto decimal     Locale.FRENCH coma decimal
    String nombre1 = sc.next();
    double nota1   = sc.nextDouble();
    String nombre2 = sc.next();
    double nota2   = sc.nextDouble();
    System.out.println(nombre1 + ": " + nota1);      // Pepe: 7.2
    System.out.println(nombre2 + ": " + nota2);      // María: 8.5
} catch (InputMismatchException e) {
    System.out.println("Error al extraer la nota");
} catch (NoSuchElementException e) {
    System.out.println("Error al extraer el dato");
}
```

La clase java.util.Scanner

- Por defecto, los delimitadores para **tokens** son espacios, tabuladores y nueva línea (`"[ \t\n\r\f]+"`), pero se pueden establecer otros (no afecta a `nextLine()`):
  - `Scanner useDelimiter(String regex);  // delimitadores para TOKENS`
    - Los delimitadores son expresiones regulares. Por ejemplo:
      - `"[,;:]"  // Exactamente uno de entre ,;:`
      - `"[,;:]+"  // Uno o más de entre ,;:`
      - `"[^a-zA-Z0-9]"  // Cualquier carácter que no sea una letra o dígito`
      - `"[,;:] ?"  // Uno de entre ,;: seguido por un espacio opcional`
      - `"\\s*[,;:]\\s*" // Uno de entre ,;: entre múltiples [ \t\n\r\f] opcionales`

```
try (Scanner sc = new Scanner("Pepe Luis : 20 ; María Luisa : 30")) {
    sc.useDelimiter("\\s*[,;:]\\s*");
    String nombre1 = sc.next();
    int    edad1   = sc.nextInt();
    String nombre2 = sc.next();
    int    edad2   = sc.nextInt();
    System.out.println(nombre1 + ": " + edad1);      // Pepe Luis: 20
    System.out.println(nombre2 + ": " + edad2);      // María Luisa: 30
} catch (InputMismatchException e) {
    System.out.println("Error al extraer la edad");
} catch (NoSuchElementException e) {
    System.out.println("Error al extraer el dato");
}
```

La clase java.util.Scanner

- Ejemplo de entrada de tokens con otros delimitadores:

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        try (Scanner sc = new Scanner("hola a ; todos. como-estas")) {
            // Delimitadores: espacio . , ; - una o más veces (+)
            sc.useDelimiter("[ .,;-]+");
            while (sc.hasNext()) {
                String cad = sc.next();
                System.out.println("[" + cad + "]");
            }
            // muestra los tokens: [hola] [a] [todos] [como] [estas]
        }
    }
}
```

La clase java.util.Scanner

- Ejemplo de **datos organizados en dos niveles** por bloques.
  1. Los datos de las personas se especifican en bloques separados por el símbolo "."
  2. Cada bloque contiene el nombre y la edad de una persona separados por uno de ",:-"

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        String datos = "Juan García, 23. Pedro González: 15."
            + "Luisa López - 19. Andrés Molina-22";
        try (Scanner sc1 = new Scanner(datos)) {
            sc1.useDelimiter("\\s*[.]\\s*");
            while (sc1.hasNext()) {
                String bloque = sc1.next();
                try (Scanner sc2 = new Scanner(bloque)) {
                    sc2.useDelimiter("\\s*[,:-]\\s*");          // Juan García: 23
                    String nombre = sc2.next();                 // Pedro González: 15
                    int edad      = sc2.nextInt();              // Luisa López: 19
                    System.out.println(nombre + ": " + edad);   // Andrés Molina: 22
                } catch (NoSuchElementException e) {
                    System.out.println("Error al extraer el dato");
                }
            }
        }
    }
}
```

La clase java.util.Scanner

- Ejemplo de **datos organizados en dos niveles** por **líneas**.
  1. Los datos de las personas se especifican en líneas (separados por el símbolo "\n")
  2. Cada línea contiene el nombre y la edad de una persona separados por uno de ",:-"

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        String datos = "Juan García, 23\nPedro González: 15\n"
            + "Luisa López - 19\nAndrés Molina-22\n";
        try (Scanner sc1 = new Scanner(datos)) {

            while (sc1.hasNextLine()) {
                String linea = sc1.nextLine();
                try (Scanner sc2 = new Scanner(linea)) {
                    sc2.useDelimiter("\\s*[,:-]\\s*");          // Juan García: 23
                    String nombre = sc2.next();                 // Pedro González: 15
                    int edad      = sc2.nextInt();              // Luisa López: 19
                    System.out.println(nombre + ": " + edad);   // Andrés Molina: 22
                } catch (NoSuchElementException e) {
                    System.out.println("Error al extraer el dato");
                }
            }
        }
    }
}
```

La clase `java.util.Scanner`

- La clase `Scanner` permite crear objetos *scanner* asociados a objetos de tipo `String`. En este caso, la **funcionalidad es similar** al método `split()` de la clase `String`. Sin embargo, existen algunas diferencias:
  - En el caso de `Scanner`, si se extrae un elemento que **no existe**, se lanza la excepción `NoSuchElementException`, mientras que el acceso a un elemento que no existe del array lanza la excepción `IndexOutOfBoundsException`.
  - En el caso de `Scanner`, si hay un **error de conversión** cuando se extrae un elemento, se lanza la excepción `InputMismatchException`, mientras que el error de conversión de `Integer.parseInt()` y `Double.parseDouble()` lanza la excepción `NumberFormatException`.
  - En el caso de `Scanner`, la extración de números reales depende de la **localización** del objeto (*británica*, *francesa*, etc) para considerar la coma o el punto decimal, mientras que el método `Double.parseDouble()` siempre utiliza el punto decimal.
  - El método `next()` de `Scanner` salta los **delimitadores iniciales** y devuelve el siguiente *token*, y este hecho afecta en el caso de que la cadena comience por un delimitador, en cuyo caso, el primer token devuelto por `split()` es un token vacío.

La clase java.util.Scanner

- Podemos crear un objeto *scanner* asociado al **teclado** (`System.in`). Así, podremos extraer datos introducidos desde el teclado.
  - Al utilizar la sentencia **"try-con-recursos"**, el objeto *scanner* será cerrado automáticamente. En este caso, el **teclado** (`System.in`) también será cerrado automáticamente, y **no podrá** ser utilizado posteriormente.

```
import java.util.Scanner;
class EjScanner {
    public static void main(String[] args) {
        try (Scanner teclado = new Scanner(System.in)) {
            System.out.print("Introduzca su nombre: ");
            String nombre = teclado.next();
            System.out.print("Introduzca su edad: ");
            int edad = teclado.nextInt();
            System.out.println(nombre + ": " + edad);
        } catch (InputMismatchException e) {
            System.out.println("Error al extraer la edad");
        } catch (NoSuchElementException e) {
            System.out.println("Error al extraer el dato");
        }
        // En este punto, el teclado (System.in) será cerrado
        // y no se podrá utilizar posteriormente
    }
}
```

La clase java.util.Scanner

- Podemos crear un objeto *scanner* asociado al **teclado** (`System.in`). Así, podremos extraer datos introducidos desde el teclado.
  - Podemos comprobar si los datos son correctos, y desechar los erróneos.

```
import java.util.Scanner;
class EjScanner {
    public static void main(String[] args) {
        try (Scanner teclado = new Scanner(System.in)) {
            System.out.print("Introduzca su nombre: ");
            String nombre = teclado.next();
            System.out.print("Introduzca su edad: ");
            while ( ! teclado.hasNextInt() ) {
                teclado.next();                    // descartamos la entrada
                System.out.print("Introduzca su edad de nuevo: ");
            }
            int edad = teclado.nextInt();
            System.out.println(nombre + ": " + edad);
        } catch (InputMismatchException e) {
            System.out.println("Error al extraer la edad");
        } catch (NoSuchElementException e) {
            System.out.println("Error al extraer el dato");
        }
        // En este punto, el teclado (System.in) será cerrado
        // y no se podrá utilizar posteriormente
    }
```

La clase java.util.Scanner

- Realizar la entrada de datos orientada a **línea** entremezclada con los otros métodos orientados a **tokens**, en el mismo objeto *scanner*, causa **problemas**.
- El método `skip()` permite eliminar elementos del flujo de entrada:
  - `Scanner skip(String regex); // salta y elimina los caracteres según regex`

```
import java.util.Scanner;
class EjScanner {
    public static void main(String[] args) {
        try (Scanner teclado = new Scanner(System.in)) {
            for (int i = 0; i < 5; ++i) {
                System.out.print("Introduzca su nombre: ");
                teclado.skip("\\s*");  // elimina todos los espacios y nl
                String nombre = teclado.nextLine();
                System.out.print("Introduzca su edad: ");
                int edad = teclado.nextInt();
                teclado.skip(".*\n");  // elimina todo hasta nueva línea
                System.out.println(nombre + ": " + edad);
            }
        } catch (NoSuchElementException e) {
            System.out.println("Error al extraer el dato");
        }
        // En este punto, el teclado (System.in) será cerrado
        // y no se podrá utilizar posteriormente
    }
}
```

Entrada/Salida. El paquete `java.io`

- La entrada y salida de datos se refiere a la **transferencia de datos** entre un programa y los dispositivos:
  - de almacenamiento (ej. disco, pendrive).
  - de comunicación.
  - con humanos (ej. teclado, pantalla, impresora).
  - con otros sistemas (ej. tarjeta de red, router).
- La **entrada** (lectura/cargar) se refiere a los datos que recibe el programa y la **salida** (escritura/guardar) se refiere a los datos que el programa envía.
  - Ya hemos visto la entrada de teclado y la salida a pantalla.
  - Ahora vamos a tratar la entrada/salida con **ficheros**.
- El paquete `java.io` se compone de una serie de interfaces, clases y excepciones destinadas a definir y controlar:
  - El sistema de ficheros.
  - Los distintos tipos de flujos de entrada y salida.
  - Las serializaciones de objetos.

Ficheros

- Los ficheros (archivos) permiten **almacenar la información** de forma **permanente** en el sistema de almacenamiento.
- Un fichero contiene cierta información **codificada**, que se almacena en una memoria como una **secuencia de bytes**.
- Cada fichero recibe un **nombre** (posiblemente con una extensión) y se ubica dentro de un **directorio** (carpeta) que forma parte de una cierta **jerarquía** (ruta, camino o vía de acceso).
- El **nombre** y la **ruta**, o secuencia de directorios, que hay que atravesar para llegar a la ubicación de un fichero, **identifican** a dicho fichero de forma única dentro del sistema de ficheros.
  - Se utilizará el símbolo / para separar los componentes de la ruta del fichero (Unix, MacOSX y Windows).
  - En caso de utilizar la *barra invertida* como carácter separador en cadenas de caracteres, entonces se debe duplicar (\\), ya que representa el símbolo de escape (Windows).

Operaciones con ficheros

- **Apertura**: establece una *conexión* entre un objeto (manejador del fichero) dentro del programa y un determinado fichero dentro del sistema de almacenamiento. En caso de apertura para entrada, el fichero debe existir. En caso de apertura para salida, se borrará o creará un nuevo fichero.
- **Escritura**: para poder almacenar información en un fichero, una vez abierto en modo de escritura, hay que transferir la información *organizada* de alguna forma mediante operaciones de escritura.
- **Lectura**: para poder utilizar la información contenida en un fichero, debe estar abierto en modo de lectura, y hay que utilizar las operaciones de lectura adecuadas a la *organización* de la información contenida en dicho fichero.
- **Cierre**: cuando se ha terminado de transferir la información a o desde el fichero, se debe *cerrar* la conexión previamente establecida entre la variable manejador del fichero y el fichero en el sistema de almacenamiento. Esta operación se ocupa además de mantener la integridad del fichero, escribiendo previamente la información que se encuentre en algún buffer intermedio en espera de pasar al fichero.

La clase java.nio.file.Files

- La clase `java.nio.file.Path` permite especificar el nombre de un fichero o directorio en la jerarquía del sistema de ficheros:

```
Path pathAbsoluto = Path.of("/dir1/dir2/nombre.txt");  // absoluto desde la raíz del sistema
Path pathRelativo = Path.of("./dir1/dir2/nombre.txt"); // relativo desde la raíz del proyecto
Path pathRelativo = Path.of("nombre.txt");             // relativo desde la raíz del proyecto
```

- La clase `java.nio.file.Files` proporciona múltiples métodos de clase que nos permiten manipular ficheros y directorios en la jerarquía del sistema de ficheros:

```
▸ boolean exists(Path);          // comprueba si existe el directorio o fichero
▸ boolean isDirectory(Path);     // comprueba si es un directorio
▸ boolean isRegularFile(Path);   // comprueba si es un fichero
▸ boolean isReadable(Path);      // comprueba si el fichero se puede leer
▸ boolean isWritable(Path);      // comprueba si el fichero se puede escribir
▸ Path createDirectory(Path);    // crea el directorio especificado
▸ Path createDirectories(Path); // crea la jerarquía de directorios especificada
▸ Path createFile(Path);         // crea el fichero especificado
▸ void delete(Path);             // elimina el directorio o fichero especificado
▸ boolean deleteIfExists(Path); // elimina el fichero o directorio especificado
▸ Path copy(Path origen, Path destino);   // copia el fichero origen a destino
▸ Path move(Path origen, Path destino);   // mueve el fichero origen a destino

▸ BufferedReader newBufferedReader(Path); // crea un manejador para leer del fichero
```

Lectura de fichero con BufferedReader

- El método de clase `newBufferedReader`, de la clase `Files` del paquete `java.nio.file`, nos permite crear un objeto de la clase `BufferedReader`, del paquete `java.io`, para poder leer y procesar las líneas de un fichero, especificado como un *path* (`Path.of("nombre.txt")`).
- Se debe utilizar dentro de la sentencia **try-con-recursos**, para que se cierre adecuadamente, incluso en presencia de excepciones.
- Puede lanzar `IOException` en caso de errores en la lectura del fichero.
- El método `readLine()`, de `BufferedReader`, lee y devuelve un `String` conteniendo la siguiente línea leída del fichero. En caso de **fin-de-fichero**, devuelve **null**. Usualmente, se leerán todas las líneas del fichero utilizando un **bucle de lectura adelantada**.

```
import java.nio.file.Path;
import java.nio.file.Files;
import java.io.BufferedReader;
import java.io.IOException;
public class Ejemplo {
    public static void leerPalabras(String nombreFichero) throws IOException {
        try (BufferedReader buffReader = Files.newBufferedReader(Path.of(nombreFichero))) {
            String linea = buffReader.readLine();
            // ...
        }
    }
}
```

Lectura de fichero con BufferedReader

Ejemplo, lectura de fichero y mostrar las palabras que estarán separadas indistintamente por los delimitadores `"\\s*[,;:.]\\s*"`.

```
import java.nio.file.Path;
import java.nio.file.Files;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
public class Ejemplo {
    public static void leerPalabras(String nombreFichero) throws IOException {
        try (BufferedReader buffReader = Files.newBufferedReader(Path.of(nombreFichero))) {
            String linea = buffReader.readLine();
            while (linea != null) {
                procesarLinea(linea);
                linea = buffReader.readLine();
            }
        }
    }
    private static void procesarLinea(String linea) {
        String[] datos = linea.split("\\s*[,;:.]\\s*");
        for (String palabra : datos) {
            System.out.println(palabra);
        }
    }
}
```

```
private static void procesarLinea(String linea) {
    try (Scanner sc = new Scanner(linea)) {
        sc.useDelimiter("\\s*[,;:.]\\s*");
        while (sc.hasNext()) {
            String palabra = sc.next();
            System.out.println(palabra);
        }
    }
}
```

Lectura de fichero con BufferedReader

- Leer de un fichero, donde cada línea contiene el nombre y la nota de un alumno, separados indistintamente por los delimitadores `"\\s*[,;:.]\\s*"`. En caso de error, desechar la línea errónea y seguir procesando las líneas siguientes.
- Atención al `try-catch` dentro de `procesarLinea` para capturar los errores y poder continuar el procesamiento con la siguiente línea.

```java
// imports ...
public class Ejemplo {
    public static void leerAlumnos(String nombreFichero) throws IOException {
        try (BufferedReader buffReader = Files.newBufferedReader(Path.of(nombreFichero))) {
            String linea = buffReader.readLine();
            while (linea != null) {
                procesarLinea(linea);
                linea = buffReader.readLine();
            }
        }
    }
    private static void procesarLinea(String linea) {
        try {
            String[] datos = linea.split("\\s*[,;:.]\\s*");
            String nombre = datos[0];
            double nota = Double.parseDouble(datos[1]);
            System.out.println(nombre + ": " + nota);
        } catch (Exception e) {
            // desechar la línea y continuar el procesamiento
        }
    }
}
```

```java
private static void procesarLinea(String linea) {
    try (Scanner sc = new Scanner(linea)) {
        sc.useDelimiter("\\s*[,;:.]\\s*");
        sc.useLocale(Locale.ENGLISH);
        String nombre = sc.next();
        double nota = sc.nextDouble();
        System.out.println(nombre + ": " + nota);
    } catch (Exception e) {
        // desechar la línea y continuar el procesamie
    }
}
```

Lectura de fichero con Scanner

1. Crear un objeto `java.nio.file.Path` con un nombre de fichero y crear un objeto `Scanner` sobre el objeto `Path` creado (puede lanzar `IOException`). Se recomienda crearlo dentro de una sentencia **"try-con-recursos"**.

```
try (Scanner sc = new Scanner(Path.of("datos.txt"))) {
    // ...
}
```

2. Leer la información con los métodos de la clase `Scanner` las veces que sean necesarias (`hasNextLine()` y `nextLine()` o `hasNext()` y `next()`).

```
while (sc.hasNextLine()) {
    String linea = sc.nextLine();
    // ... procesar la información
}
```

```
while (sc.hasNext()) {
    String token = sc.next();
    // ... procesar la información
}
```

3. En el caso de que el fichero esté organizado en **líneas**, se aconseja realizar la lectura **en dos niveles**. En un primer nivel se leen las líneas con `nextLine()`, y en un segundo nivel, se extrae la información de la línea y se procesa.

4. Cerrar el objeto `Scanner`, considerando que si el objeto se crea dentro de una sentencia **"try-con-recursos"**, entonces se cierra automáticamente, y no es necesario invocar explícitamente a `sc.close()`.

   En caso de **no cerrar adecuadamente** el objeto `Scanner`, entonces es posible que se **pierdan** *recursos del sistema*.

Lectura de fichero con Scanner

Ejemplo, lectura de fichero y mostrar las palabras que estarán separadas indistintamente por los delimitadores `"\\s*[,;:.]\\s*"`.

```
import java.nio.file.Path;
import java.io.IOException;
import java.util.Scanner;
public class Ejemplo {
    public static void leerPalabras(String nombreFichero) throws IOException {
        try (Scanner sc = new Scanner(Path.of(nombreFichero))) {
            while (sc.hasNextLine()) {
                String linea = sc.nextLine();
                procesarLinea(linea);
            }
        }
    }
    private static void procesarLinea(String linea) {
        try (Scanner sc = new Scanner(linea)) {
            sc.useDelimiter("\\s*[,;:.]\\s*");
            while (sc.hasNext()) {
                String palabra = sc.next();
                System.out.println(palabra);
            }
        }
    }
```

```
private static void procesarLinea(String linea) {
    String[] datos = linea.split("\\s*[,;:.]\\s*");
    for (String palabra : datos) {
        System.out.println(palabra);
    }
}
```

Lectura de fichero con Scanner

- Leer de un fichero, organizado en líneas, donde cada línea contiene el nombre y la nota de un alumno, separados indistintamente por los delimitadores `"\\s*[,;:.]\\s*"`. En caso de error, desechar la línea errónea y seguir procesando las líneas siguientes.
- Atención al `try-catch` dentro de `procesarLinea` para capturar los errores y poder continuar el procesamiento con la siguiente línea.

```java
// imports ...
public class Ejemplo {
    public static void leerAlumnos(String nombreFichero) throws IOException {
        try (Scanner sc = new Scanner(Path.of(nombreFichero))) {
            while (sc.hasNextLine()) {
                String linea = sc.nextLine();
                procesarLinea(linea);
            }
        }
    }
    private static void procesarLinea(String linea) {
        try (Scanner sc = new Scanner(linea)) {
            sc.useDelimiter("\\s*[,;:.]\\s*");
            sc.useLocale(Locale.ENGLISH);
            String nombre = sc.next();
            double nota = sc.nextDouble();
            System.out.println(nombre + ": " + nota);
        } catch (Exception e) {
            // desechar la línea y continuar el procesamiento
        }
    }
```

```java
private static void procesarLinea(String linea) {
    try {
        String[] datos = linea.split("\\s*[,;:.]\\s*");
        String nombre = datos[0];
        double nota = Double.parseDouble(datos[1]);
        System.out.println(nombre + ": " + nota);
    } catch (Exception e) {
        // desechar la línea y continuar el procesamient
    }
}
```

La clase java.io.PrintWriter

- Los objetos de la clase `PrintWriter` permiten escribir objetos y tipos primitivos sobre flujos de salida de texto.
- Constructores:
  - `PrintWriter(String nombreFichero); // incluye la ruta completa`
  - `PrintWriter(FileWriter out);  // salida a Fichero-modo-append`
  - `PrintWriter(StringWriter out);  // salida a StringWriter -> String`
  - `PrintWriter(OutputStream out, boolean autoFlush); // (System.out, true)`
- Métodos de instancia para escribir todos los tipos básicos y objetos:
  - `void print(...);  // escribe tipos primitivos y objetos`
  - `void println(...);  // escribe tipos primitivos y objetos`
  - `void printf(String formato, ...); // escribe con formato`
  - `void printf(Locale loc, String formato, ...); // escribe con loc y formato`
  - `void flush();  // fuerza la salida de datos`
- Se debe **cerrar** el objeto `PrintWriter` cuando finaliza el procesamiento de texto.
  - `void close();`

Escritura sobre un fichero de texto

1. Crear un objeto `PrintWriter` utilizando un nombre de fichero (puede lanzar `FileNotFoundException`/`IOException`). Se recomienda crearlo dentro de una sentencia **"try-con-recursos"**.

```
PrintWriter pw = new PrintWriter("datos.txt");
```

2. Escribir la información sobre el objeto `PrintWriter` las veces que sean necesarias, atendiendo al formato del fichero.

```
pw.println("Hola a todos");
```

3. Cerrar el objeto `PrintWriter`, considerando que si el objeto se crea dentro de una sentencia **"try-con-recursos"**, entonces se cierra automáticamente, y no es necesario invocar explícitamente a `close()`.

```
pw.close();
```

En caso de **no cerrar adecuadamente** el objeto `PrintWriter`, entonces es posible que el fichero **no** guarde correctamente todos los datos enviados, y además se **pierdan** *recursos del sistema*.

Escritura sobre un fichero de texto

- Usualmente, cuando se guarda información en un fichero, se **borra** el contenido anterior y se guarda todo el contenido del fichero con la nueva información.

```
import java.io.IOException;
import java.io.PrintWriter;

public class Ejemplo {
    private static final String[] DATOS = { "hola", "como", "estás" };
    private static void imprimirDatos(PrintWriter pw) {
        for (String pal : DATOS) {
            pw.println(pal);
        }
    }
    public static void escribirPalabras(String nombreFichero) throws IOException {
        // Crear un nuevo fichero de palabras (reemplaza los datos anteriores)
        try (PrintWriter pw = new PrintWriter(nombreFichero)) {
            imprimirDatos(pw);
        }
    }
}
```

Añadir datos a un fichero de texto ya existente (Append)

- Usualmente, cuando se guarda información en un fichero, se **borra** el contenido anterior y se guarda todo el contenido del fichero con la nueva información.
- A veces interesa poder **añadir** la nueva información al final del fichero, a continuación de la información que ya estaba almacenada, sin eliminarla.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.io.FileWriter;
public class Ejemplo {
    private static final String[] DATOS = { "hola", "como", "estás" };
    private static void imprimirDatos(PrintWriter pw) {
        for (String pal : DATOS) {
            pw.println(pal);
        }
    }
    public static void escribirPalabras(String nombreFichero) throws IOException {
        // Añadir información a un fichero de palabras
        try (PrintWriter pw = new PrintWriter(new FileWriter(nombreFichero, true))) {
            imprimirDatos(pw);
        }
    }
}
```

Asociar un Objeto PrintWriter a un String

Este caso se utilizará en la programación de *GUIs*, en el *controlador* del patrón de diseño *Modelo-Vista-Controlador*, para poder reenviar la salida de datos a un `String`, y de allí, a un determinado *area de texto* de la *GUI*.

```
import java.io.StringWriter;
import java.io.PrintWriter;
public class Ejemplo {
    private static final String[] DATOS = { "hola", "como", "estás" };
    private static void imprimirDatos(PrintWriter pw) {
        for (String pal : DATOS) {
            pw.println(pal);
        }
    }
    public static String escribirPalabrasAString() {
        // Enviar información a un String a través de PrintWriter
        StringWriter salida = new StringWriter();
        try (PrintWriter pw = new PrintWriter(salida)) {
            imprimirDatos(pw);
        }
        return salida.toString();
    }
}
```

Asociar un Objeto PrintWriter a la Consola System.out

A veces interesa poder enviar a la *Consola* la salida de cierta información, que inicialmente se suele enviar a fichero.

```
import java.io.PrintWriter;
public class Ejemplo {
    private static final String[] DATOS = { "hola", "como", "estás" };
    private static void imprimirDatos(PrintWriter pw) {
        for (String pal : DATOS) {
            pw.println(pal);
        }
    }
    public static void mostrarPalabrasEnConsola() {
        // Enviar información a consola a través de PrintWriter
        // En este caso no se puede utilizar try-con-recursos
        // ni tampoco se puede cerrar el PrintWriter, ya que
        // entonces también cerraría la consola System.out
        PrintWriter pw = null;
        try {
            pw = new PrintWriter(System.out, true); // TRUE -->  auto-flush
            imprimirDatos(pw);
        } finally {
            if (pw != null) {
                pw.flush(); // flush fuerza el volcado de los datos a consola
            }
        }
    }
```