


Clases Básicas Predefinidas. Entrada/Salida

Contenido

- Organización en paquetes 
- Clases básicas: `java.lang`
- Clases útiles del paquete `java.util`
- Entrada/Salida (Ficheros). El paquete `java.io`, `java.nio` y `java.nio.file`

Organización en paquetes (packages)

Como ya vimos en el Tema 2

- Un paquete en Java es un mecanismo para agrupar clases e interfaces relacionados desde un punto de vista lógico.
- Físicamente, un paquete se corresponde con un subdirectorio o carpeta.
- Las clases de un paquete sólo se pueden acceder directamente por sus nombres desde otra clase dentro del mismo paquete.
- Para acceder a ellas desde otro paquete hay que hacerlo precediéndolas con el nombre del paquete o utilizando la construcción **import**

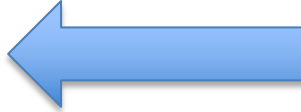
Organización en paquetes (packages)

API


(Application Programming Interface)

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

Contenido

- Organización en paquetes
- Clases básicas: `java.lang` 
- Clases útiles del paquete `java.util`
- Entrada/Salida (Ficheros). El paquete `java.io`, `java.nio` y `java.nio.file`

El paquete java.lang

- Siempre está incluido en cualquier aplicación, no es necesario importarlo explícitamente.
- Contiene las clases básicas del sistema:
 - **Object** 
 - **System**
 - **Math**
 - **String, StringBuilder y StringBuffer**
 - Envoltorios de tipos básicos
 - ...
- Contiene interfaces:
 - **Comparable**
 - **Iterable**
 - ...
- Contiene también excepciones:
 - **ArrayIndexOutOfBoundsException**
 - **NullPointerException**
 - ...

La clase Object

- Es la clase superior de toda la jerarquía de clases de Java.
 - Define el comportamiento mínimo común de todos los objetos.
 - Si una clase no hereda de otra, entonces hereda de **Object**. Todas las clases heredan de ella directa o indirectamente.
 - No es una clase abstracta pero no tiene mucho sentido crear instancias suyas.
- Métodos de instancia importantes (tienen una implementación por defecto, pero lo normal es redefinirlos):
 - **String toString()**
ya visto en Tema 2. Por defecto devuelve “nombreClase@valorHex”
 - **boolean equals(Object o)**
 - **int hashCode()**
 - ...

El método `equals()`

- Compara dos objetos de la misma clase (o subclases).
- Por defecto realiza una comparación por `==`.
- Este método se puede (y se debe) **redefinir** en cualquier clase para comparar objetos de esa clase (o subclases).

El método equals ()

- Compara dos objetos de la misma clase (o subclases).
- Por defecto realiza una comparación por ==.
- Este método se puede (y se debe) **redefinir** en cualquier clase para comparar objetos de esa clase (o subclases).

```
class Persona {  
    private String nombre;  
    private int edad;  
    public Persona(String n, int e) {  
        nombre = n;  
        edad = e;  
    }  
    public boolean equals(Object o) {  
        boolean res = false;  
        if (o instanceof Persona) {  
            Persona p = (Persona) o;  
            res = edad == p.edad &&  
                nombre.equals(p.nombre);  
        }  
        return res;  
    }  
}
```

Importante
Object


Instancia de
Persona o de
una subclase

Conversión
de Tipo

El método equals ()

- Compara dos objetos de la misma clase (o subclases).
- Por defecto realiza una comparación por ==.
- Este método se puede (y se debe) **redefinir** en cualquier clase para comparar objetos de esa clase (o subclases).

```
class Persona {  
    private String nombre;  
    private int edad;  
    public Persona(String n, int e) {  
        nombre = n;  
        edad = e;  
    }  
    public boolean equals(Object o) {  
        boolean res = false;  
        if (o instanceof Persona) {  
            Persona p = (Persona) o;  
            res = edad == ((Persona) p).edad &&  
                nombre.equals(((Persona) p).nombre);  
        }  
        return res;  
    }  
}
```




Sin
Variable
auxiliar

El método equals ()

- Compara dos objetos de la misma clase (o subclases).
- Por defecto realiza una comparación por ==.
- Este método se puede (y se debe) **redefinir** en cualquier clase para comparar objetos de esa clase (o subclases).

```
class Persona {  
    private String nombre;  
    private int edad;  
    public Persona(String n, int e) {  
        nombre = n;  
        edad = e;  
    }  
    public boolean equals(Object o) {  
        boolean res = o instanceof Persona;  
        Persona p = res ? (Persona)o : null;  
        return res && edad == p.edad && nombre.equals(p.nombre);  
    }  
}
```



Otra forma
mas corta

El método `hashCode()`

- Devuelve un `int` para cada objeto de una clase.
- Por defecto devuelve un valor relacionado con la dirección del objeto
- Para los tipos básicos existen clases que nos permitirán calcular su `hashCode`.

```
Double.hashCode(...)
```

```
Integer.hashCode(...)
```

```
...
```

- Además, la clase `Objects` de `java.util` dispone de un método de clase `hash` para calcular el `hashCode` de cualquier clase:

```
Objects.hash(...);
```

`equals()` y `hashCode()`

- Hay una relación que debe mantenerse entre `equals()` y `hashCode()`;

`a.equals(b) ==> a.hashCode() == b.hashCode()`

- Todas las clases de la API de Java verifican esa relación.

El que crea una clase es el responsable de mantener esta relación, redefiniendo adecuadamente los métodos:

- `boolean equals(Object)`
- `int hashCode()`



Fundamental para que posteriormente los objetos de esa clase puedan almacenarse correctamente en colecciones basadas en el uso de **Tablas Hash** (**Tema 5**)

`equals()` y `hashCode()`

```
class Persona {  
    private String nombre;  
    private int edad;  
    public Persona(String n, int e) {  
        nombre = n;  
        edad = e;  
    }  
    public boolean equals(Object o) {  
        boolean res = o instanceof Persona;  
        Persona p = res ? (Persona)o : null;  
        return res && edad == p.edad && nombre.equals(p.nombre);  
    }  
    public int hashCode() {  
        return Objects.hash(nombre, edad);  
    }  
}
```

El método hash de la clase `Objects` de `java.util` distribuye muy bien los valores hash.

En `hashCode()` deben intervenir las variables que se usan en `equals`

`equals()` y `hashCode()`

```
class Persona {  
    private String nombre;  
    private int edad;  
    public Persona(String n, int e) {  
        nombre = n;  
        edad = e;  
    }  
    public boolean equals(Object o) {  
        return (o instanceof Persona) &&  
            (edad == ((Persona) o).edad) &&  
            ((Persona) o).nombre.equals(nombre);  
    }  
    public int hashCode() {  
        return nombre.hashCode() + Integer.hashCode(edad);  
    }  
}
```

Podemos usar diferentes
operaciones

equals () y hashCode ()

```
class Persona {  
    private String nombre;  
    private int edad;  
    public Persona(String n, int e) {  
        nombre = n;  
        edad = e;  
    }  
    public boolean equals(Object o) {  
        return (o instanceof Persona) &&  
            (edad == ((Persona) o).edad) &&  
            ((Persona) o).nombre.equals(nombre);  
    }  
    public int hashCode() {  
        return nombre.hashCode() + edad;  
    }  
}
```


Si se trata de un entero, se puede
usar directamente

equals () y hashCode ()

```
class Persona {  
    private String nombre;  
    private int edad;  
    public Persona(String n, int e) {  
        nombre = n;  
        edad = e;  
    }  
    public boolean equals(Object o) {  
        return (o instanceof Persona) &&  
            (edad == ((Persona) o).edad) &&  
            ((Persona) o).nombre.equalsIgnoreCase(nombre);  
    }  
    public int hashCode() {  
        return Objects.hash(nombre.toLowerCase(), edad);  
    }  
}
```

Si se usa equalsIgnoreCase para los String en equals, en hashCode debe usarse toLowerCase o toUpperCase


El paquete java.lang

- Siempre está incluido en cualquier aplicación, no es necesario importarlo explícitamente.
- Contiene las clases básicas del sistema:
 - **Object**
 - **System** 
 - **Math**
 - **String, StringBuilder y StringBuffer**
 - Envoltorios de tipos básicos
 - ...
- Contiene interfaces:
 - **Comparable**
 - **Iterable**
 - ...
- Contiene también excepciones:
 - **ArrayIndexOutOfBoundsException**
 - **NullPointerException**
 - ...

La clase `System`

- Maneja particularidades del sistema.
- Contiene variables de clase (estáticas) públicas:
 - `PrintStream out, err`
 - `InputStream in`
- Contiene métodos de clase (estáticos) públicos:
 - `void arrayCopy(...)`
 - `long currentTimeMillis()`
 - `void gc()`
 - ...

El paquete java.lang

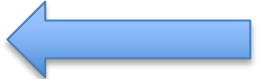
- Siempre está incluido en cualquier aplicación, no es necesario importarlo explícitamente.
- Contiene las clases básicas del sistema:
 - **Object**
 - **System**
 - **Math** 
 - **String, StringBuilder y StringBuffer**
 - Envoltorios de tipos básicos
 - ...
- Contiene interfaces:
 - **Comparable**
 - **Iterable**
 - ...
- Contiene también excepciones:
 - **ArrayIndexOutOfBoundsException**
 - **NullPointerException**
 - ...

La clase **Math**

- Incorpora como *variables y métodos de clase* (estáticos) constantes y funciones matemáticas:
 - Constantes
 - `double E`, `double PI`
 - Métodos :

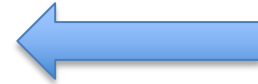
```
double sin(double), double cos(double),  
double tan(double), double asin(double),  
double acos(double), double atan(double),...  
xxx abs(xxx), xxx max(xxx,xxx), xxx min(xxx,xxx),  
double exp(double), double pow(double, double),  
double sqrt(double), int round(float),...
```
 - Consultar la documentación de la API para información adicional

El paquete java.lang

- Siempre está incluido en cualquier aplicación, no es necesario importarlo explícitamente.
- Contiene las clases básicas del sistema:
 - **Object**
 - **System**
 - **Math**
 - **String, StringBuilder y StringBuffer** 
 - Envoltorios de tipos básicos
 - ...
- Contiene interfaces:
 - **Comparable**
 - **Iterable**
 - ...
- Contiene también excepciones:
 - **ArrayIndexOutOfBoundsException**
 - **NullPointerException**
 - ...

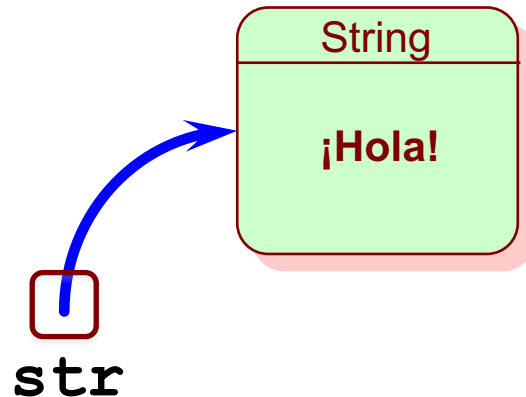
Cadenas de caracteres

- Las cadenas de caracteres se representan en Java como secuencias de caracteres Unicode encerradas entre comillas dobles.
- Para manipular cadenas de caracteres se utilizan tres clases incluidas en `java.lang`:
 - **String** - para cadenas inmutables
 - **StringBuilder** - para cadenas modificables
 - **StringBuffer** - para cadenas modificables (seguras ante hebras)



La clase **String**

- Ya vimos algunas características en el tema 2
- Cada objeto de tipo **String** alberga una cadena de caracteres.
- Las variables de esta clase se pueden inicializar:
 - de la forma normal:
`String str = new String("¡Hola!");`
 - de la forma simplificada:
`String str = "¡Hola!";`



La clase **String**

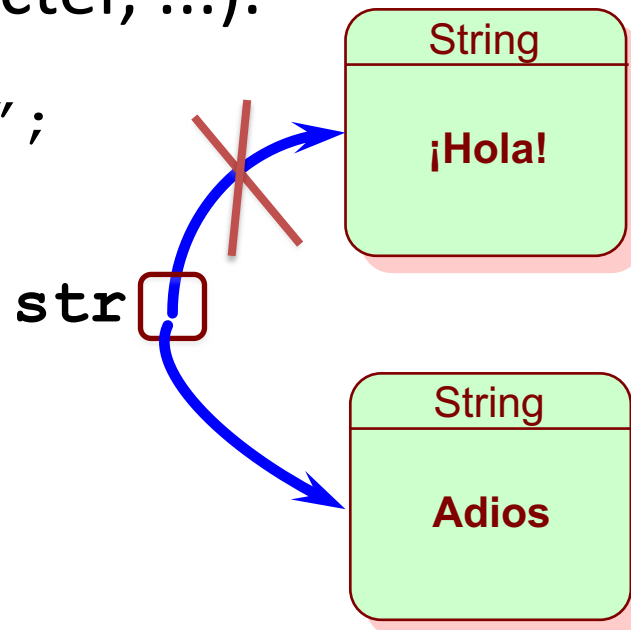
- Ya vimos algunas características en el tema 2
- A una **variable** de tipo **String** se le puede **asignar cadenas distintas** durante su existencia.
- Pero **una cadena de caracteres** almacenada en un objeto de tipo **String** **NO puede modificarse** (crecer, cambiar un carácter, ...).

```
String str = "¡Hola!";
```

```
...
```

```
str = "Adios";
```

```
...
```



Métodos de la clase `String`

- Métodos de consulta:

```
int length()
```

```
char charAt(int pos)
```

```
int indexOf/lastIndexOf(char car)
```

```
int indexOf/lastIndexOf(String str)
```

```
int indexOf/lastIndexOf(char car, int desde)
```

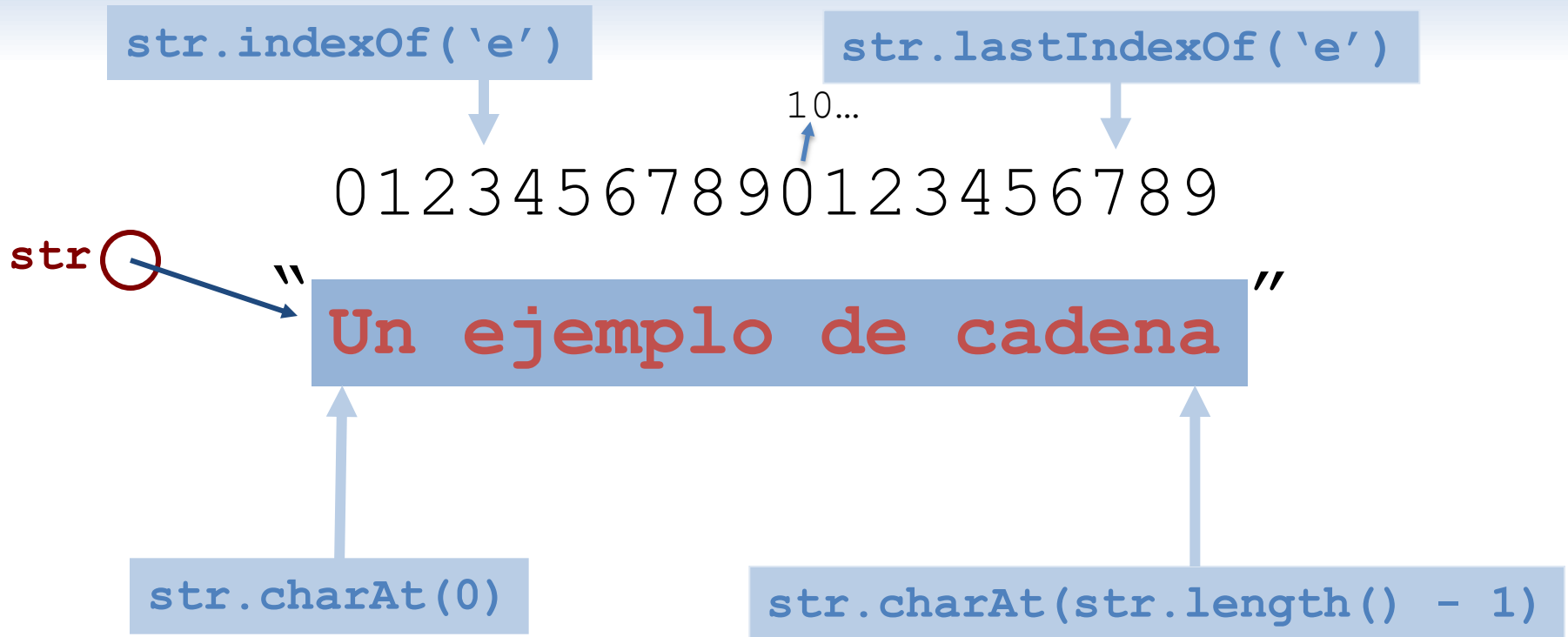
```
int indexOf/lastIndexOf(String str, int desde)
```

...

- Si se intenta acceder a una posición no válida el sistema lanza una excepción:

`StringIndexOutOfBoundsException`

Ejemplo



Métodos de la clase `String`

- Comparación de cadenas (`String c1, c2`):
 - `c1.equals(c2)` – devuelve `true` si `c1` y `c2` son iguales y `false` en otro caso.
 - `c1.equalsIgnoreCase(c2)` – Igual que la anterior, pero sin tener en cuenta las diferencias por mayúsculas y minúsculas.
 - `c1.compareTo(c2)` – devuelve un entero menor, igual o mayor que cero cuando `c1` es menor, igual o mayor que `c2`.
 - `c1.compareToIgnoreCase(c2)` – Igual que la anterior, pero sin tener en cuenta las diferencias por mayúsculas y minúsculas.
- ¡ojo!
 - `c1 == c2`, `c1 != c2`, ... (operadores relacionales) comparan variables referencia

Métodos de la clase `String`

- Métodos que producen nuevos objetos `String`:

`String substring(int posini, int posfin)`

`String substring(int posini)`

```
String str1 = "Antonio Ruiz Moreno";  
String str2;  
  
str2 = str1.substring(8,12); // str2 será "Ruiz"
```

```
String str1 = "Antonio Ruiz Moreno";  
String str2;  
  
str2 = str1.substring(8);    // str2 será "Ruiz Moreno"
```

- Si se intenta acceder a una posición no válida el sistema lanza una excepción:

`StringIndexOutOfBoundsException`

Métodos de la clase `String`

- Métodos que producen nuevos objetos `String`:

`String replace(String str1, String str2)`

`String replaceFirst(String str1, String str2)`

```
String str1 = "Antonio Ruiz Ruiz";  
String str2;  
  
str2 = str1.replace("Ruiz","Rubio");  
           // str2 será "Antonio Rubio Rubio"
```

```
String str1 = "Antonio Ruiz Ruiz";  
String str2;  
  
str2 = str1.replaceFirst("Ruiz","Rubio");  
           // str2 será "Antonio Rubio Ruiz"
```

Métodos de la clase `String`

- Métodos que producen nuevos objetos `String`:

`String concat(String s)` // también con `+`

```
String str1 = "Antonio";  
String str2 = " Luis";  
String str3 = str1.concat(str2);  
// str3 será "Antonio Luis"
```

```
String str1 = "Antonio";  
String str2 = " Luis";  
String str3 = str1 + str2;  
// str3 será "Antonio Luis"
```

Métodos de la clase `String`

- Métodos que producen nuevos objetos `String`:

`String toUpperCase()`

`String toLowerCase()`

```
String str1 = "Antonio";  
String str2;  
  
str2 = str1.toUpperCase(); // str2 será "ANTONIO"
```

```
String str1 = "Antonio";  
String str2;  
  
str2 = str1.toLowerCase(); // str2 será "antonio"
```


Métodos de la clase `String`

- Métodos que producen nuevos objetos `String`:

`static String format(String formato,...)`

- Permite construir salidas con formato.

```
String ej = "Cadena de ejemplo";  
String s = String.format("La cadena %s mide %d", ej, ej.length());  
System.out.println(s);
```

`La cadena Cadena de ejemplo mide 17`

- Formatos más comunes (se aplican con %):
 - `s` para cualquier objeto. Se aplica `toString()`. `"%20s"`
 - `d` para números sin decimales. `"%7d"`
 - `f` para números con decimales. `"%9.2f"`
 - `b` para booleanos `"%b"`
 - `c` para caracteres. `"%c"`
- Se pueden producir las excepciones:
 - `MissingFormatArgumentException`
 - `IllegalFormatConversionException`
 - `UnknownFormatConversionException`
 - ...

Métodos de la clase `String`

- Métodos que producen nuevos objetos `String`:

`static String format(String formato,...)`

- Si el resultado de `format` es para mostrarlo por pantalla, podemos utilizar directamente `printf(String formato, ...)`:

```
String ej = "Cadena de ejemplo";  
System.out.printf("La cadena %s mide %d\n", ej, ej.length());
```

`La cadena Cadena de ejemplo mide 17`

Métodos de la clase `String`

Permite extraer datos de una cadena según unos delimitadores:

```
String [] split(String exprReg)
```

Ejemplos de delimitadores:

`" [, . ; :] "` El delimitador es una aparición de espacio, coma, punto, punto y coma o dos puntos:

```
String str = ":juan garcia;17..,carpintero";
```

```
String [] items = str.split("[ , . ; : ]");
```

```
items->{"", "juan", "garcia", "17", "", "", "carpintero"}
```


`" [, . ; :] + "` El delimitador es una aparición de uno o mas símbolos de entre coma, punto, punto y coma o dos puntos:

```
String str = ":juan garcia;17..,carpintero";
```

```
String [] items = str.split("[ , . ; : ]+");
```

```
items->{"", "juan garcia", "17", "carpintero"}
```

Cadenas de caracteres

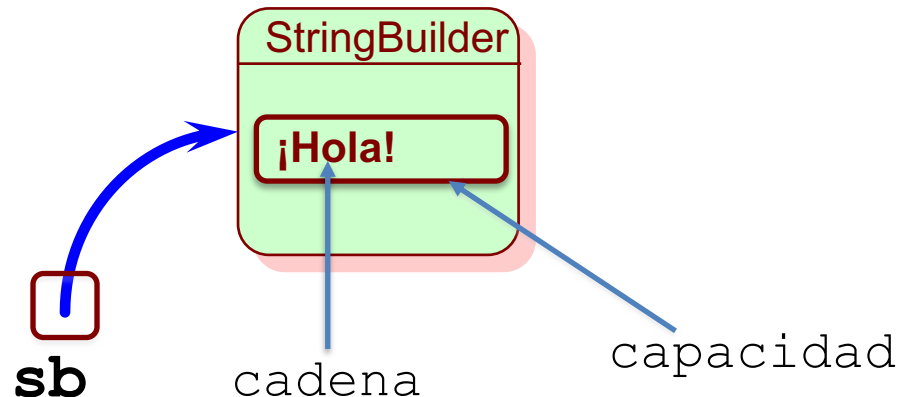
- Las cadenas de caracteres se representan en Java como secuencias de caracteres Unicode encerradas entre comillas dobles.
- Para manipular cadenas de caracteres se utilizan tres clases incluidas en `java.lang`:
 - **String** - para cadenas inmutables
 - **StringBuilder** - para cadenas modificables 
 - **StringBuffer** - para cadenas modificables (seguras ante hebras)

La clase `StringBuilder`

- Al igual que un objeto de tipo `String`, cada objeto de tipo `StringBuilder` alberga una cadena de caracteres.
- Un objeto de tipo `StringBuilder` además tiene una determinada capacidad de almacenamiento (igual o mayor que el número de caracteres de la cadena), cuyo valor también puede consultarse.
- Cuando la capacidad de almacenamiento establecida se excede, se aumenta automáticamente.

```
StringBuilder sb = new StringBuilder("¡Hola!");
```

...



La clase `StringBuilder`

- Los objetos de esta clase se inicializan de cualquiera de las formas siguientes (**no se puede asignar la cadena directamente**):

```
StringBuilder sb = new StringBuilder();
```

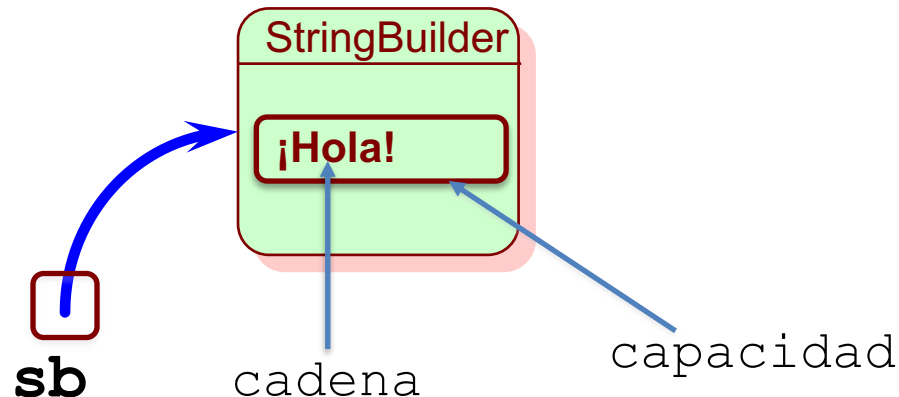
```
// cadena vacía "" y capacidad inicial para 16 caracteres
```

```
StringBuilder sb = new StringBuilder(10);
```

```
// cadena vacía "" y capacidad inicial para 10 caracteres
```

```
StringBuilder sb = new StringBuilder("¡Hola!");
```

```
// cadena "¡Hola!" y capacidad inicial para 6+16 caracteres
```



La clase **StringBuilder**

- Al igual que a una variable de tipo **String**, a una variable de tipo **StringBuilder** se le puede asignar cadenas distintas durante su existencia.
- Al contrario que ocurre con el tipo **String**, una cadena de caracteres almacenada en un objeto de tipo **StringBuilder** **SÍ** se puede ampliar, reducir y modificar mediante las funciones correspondientes.

Métodos de la clase `StringBuilder`

- Métodos de consulta:

```
int length()
```

```
int capacity()
```

```
char charAt(int pos)
```

```
int indexOf/lastIndexOf(String str)
```

```
int indexOf/lastIndexOf(String str, int desde)
```

```
...
```

- Si se intenta acceder a una posición no válida el sistema lanza una excepción:

```
StringIndexOutOfBoundsException
```


Métodos de la clase `StringBuilder`

- Métodos para construir objetos `String`:
`String substring(int posini, int posfin)`
`String substring(int posini)`
`String toString()`
...
- Si se intenta acceder a una posición no válida el sistema lanza una excepción:
`StringIndexOutOfBoundsException`

Métodos de la clase **StringBuilder**

- La clase **StringBuilder** no tiene definidos los métodos para realizar comparaciones que tiene la clase **String**.
- Pero se puede usar el método **toString()** para obtener un **String** a partir de un **StringBuilder** y poder usarlo para comparar.

```
StringBuilder sb1, sb2;  
  
...  
if (sb1.toString().equals(sb2.toString())) {  
    ...  
}
```

Métodos de la clase `StringBuilder`

- Métodos para modificar objetos `StringBuilder`:

```
StringBuilder append(String str)
StringBuilder insert(int pos, String str)
void setCharAt(int pos, char car)
StringBuilder replace(int pos1, int pos2,
                     String str)
StringBuilder reverse()
...
```

```
StringBuilder sb = new StringBuilder("Hola");
...
sb.append(" Antonio"); // sb será "Hola Antonio"
sb.insert(5, "Jose "); // sb será "Hola Jose Antonio"
// sb.append(" Antonio").insert(5, "Jose ");
```

- Si se intenta acceder a una posición no válida el sistema lanza una excepción:

`StringIndexOutOfBoundsException`

Métodos de la clase `StringBuilder`

- Métodos para modificar objetos `StringBuilder`:

```
StringBuilder append(String str)
StringBuilder insert(int pos, String str)
void setCharAt(int pos, char car)
StringBuilder replace(int pos1, int pos2,
                     String str)
StringBuilder reverse()
...
```

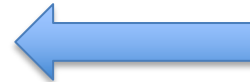
```
StringBuilder sb = new StringBuilder("esto es un ejemplo");
...
sb.setCharAt(0, 'E');           // sb será "Esto es un ejemplo"
sb.replace(8, 10, "otro");      // sb será "Esto es otro ejemplo"
```

- Si se intenta acceder a una posición no válida el sistema lanza una excepción:

`StringIndexOutOfBoundsException`

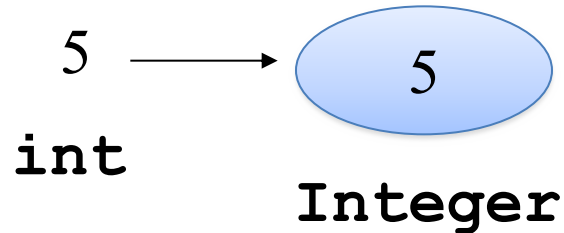
El paquete java.lang

- Siempre está incluido en cualquier aplicación, no es necesario importarlo explícitamente.
- Contiene las clases básicas del sistema:
 - **Object**
 - **System**
 - **Math**
 - **String, StringBuilder y StringBuffer**
 - Envoltorios de tipos básicos
 - ...
- Contiene interfaces:
 - **Comparable**
 - **Iterable**
 - ...
- Contiene también excepciones:
 - **ArrayIndexOutOfBoundsException**
 - **NullPointerException**
 - ...



Las clases envoltorios (*wrappers*)

- Ya sabemos que los valores de los tipos básicos no son objetos
 - Una variable de objeto almacena una referencia al objeto
 - Una variable de tipo básico almacena el valor en sí
- A veces es útil manejar valores de tipos básicos como si fueran objetos (por ejemplo para almacenarlos en colecciones (Tema 5))
- Para ello se utilizan las clases envoltorios
- Los objetos de las clases envoltorios son inmutables
- A partir de JDK1.5 se envuelve y desenvuelve automáticamente



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

Los envoltorios numéricos

```
List<Integer> lista = new ArrayList<>();  
lista.add(5); // se guarda un objeto de la clase Integer que contiene un 5.
```

- Métodos de clase para crear números a partir de cadenas de caracteres:

```
static xxxx parseXxxx(String)  
  
    int         i = Integer.parseInt("234");  
    double      d = Double.parseDouble("34.67");
```

- Métodos de clase para crear envoltorios de números a partir de cadenas de caracteres:

```
static Xxxx valueOf(String)  
  
    Integer oi = Integer.valueOf("234");  
    Double od = Double.valueOf("34.67");
```

- Se lanzan excepciones (**NumberFormatException**) si los datos no son correctos

Los envoltorios numéricos

- Métodos de clase para comparar datos básicos:

```
static int compare(xxxx v1, xxxx v2)  
    int r1 = Integer.compare(45, 78);      // r1 < 0  
    int r2 = Double.compare(34.25, 21.45);  // r2 > 0  
    int r3 = Long.compare(45, 45);          // r3 == 0
```

- Métodos de clase para calcular el hashCode de datos básicos:

```
static int hashCode(xxxx v)  
    int r1 = Integer.hashCode(45);  
    int r2 = Double.hashCode(34.25);  
    int r3 = Long.hashCode(45);
```


El envoltorio Boolean

- Método de clase para crear un valor lógico a partir de cadenas de caracteres:
static boolean parseBoolean(String)
`boolean b = Boolean.parseBoolean("true");`
- Método de clase para crear un envoltorio lógico a partir de cadenas de caracteres:
static Boolean valueOf(String)
`Boolean ob = Boolean.valueOf("false");`
- Si la cadena no representa un valor lógico no produce error, sino que lo toma como **false**

El envoltorio Boolean

- Método de clase para comparar datos lógicos:

```
static int compare(boolean v1, boolean v2)  
    int r1 = Boolean.compare(false, true);    // r1 < 0  
    int r2 = Boolean.compare(true, false);    // r2 > 0  
    int r3 = Boolean.compare(true, true);     // r3 == 0
```

- Método de clase para calcular el hashCode de datos lógicos:

```
static int hashCode(boolean v)  
    int r = Boolean.hashCode(true);
```

El envoltorio Character

- Métodos de clase para comprobar el tipo de los caracteres:

```
static boolean isDigit(char)
static boolean isLetter(char)
static boolean isLowerCase(char)
static boolean isUpperCase(char)
static boolean isSpaceChar(char)
```

```
boolean b = Character.isLowerCase('g');
```

- Métodos de clase para convertir caracteres:

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

```
char c = Character.toUpperCase('g');
```

El envoltorio Character

- Método de clase para comparar caracteres:

```
static int compare(char v1, char v2)
    int r1 = Character.compare('a', 'c');    // r1 < 0
    int r2 = Character.compare('h', 'e');    // r2 > 0
    int r3 = Character.compare('p', 'p');    // r3 == 0
```

- Método de clase para calcular el hashCode de caracteres:

```
static int hashCode(char v)
    int r = Character.hashCode('a');
```

Diferencia entre tipo básico y clase envoltorio

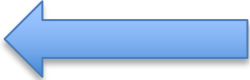
- Son datos intercambiables
- Las clases envoltorios se usan para definir el tipo del contenido en estructuras genéricas:

```
List<Integer> lista = new ArrayList<>();  
lista.add(3);  
lista.add(5 + lista.get(0));  
int i = lista.get(1);
```

- Cuidado porque la clase Integer pueden contener null (las clases en general):

```
Integer i = null;  
int j = i + 1; // NullPointerException
```

El paquete java.lang

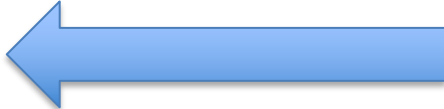
- Siempre está incluido en cualquier aplicación, no es necesario importarlo explícitamente.
- Contiene las clases básicas del sistema:
 - **Object**
 - **System**
 - **Math**
 - **String**, **StringBuilder** y **StringBuffer**
 - Envoltorios de tipos básicos
 - ...
- Contiene interfaces:  **Tema 5**
 - **Comparable**
 - **Iterable**
 - ...
- Contiene también excepciones:
 - **ArrayIndexOutOfBoundsException**
 - **NullPointerException**
 - ...

Contenido

- Organización en paquetes
- Clases básicas: `java.lang`
- Clases útiles del paquete `java.util`
- Entrada/Salida (Ficheros). El paquete `java.io`, `java.nio` y `java.nio.file`



El paquete `java.util`

- Contiene clases de utilidad
 - Las colecciones (se verán en el Tema 5)
 - La clase **Random**. 
 - La clase **StringJoiner**.
 - La clase **Scanner**.
 - Contiene también interfaces y excepciones
 - ... consultar la documentación.

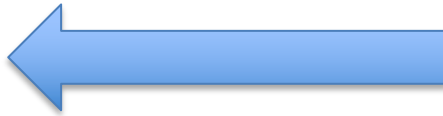
La clase Random

- Los objetos de esta clase permiten generar números aleatorios de diversas formas mediante diferentes métodos de instancia:

```
double nextDouble()  
    // número real entre 0.0 y 1.0 (no incluido)  
int nextInt()  
    // número entero entre  $-2^{32}$  y  $2^{32} - 1$   
int nextInt(int n)  
    // número entero entre 0 y n (no incluido)  
...
```




- Consultar la documentación para información adicional.

El paquete `java.util`

- Contiene clases de utilidad
 - Las colecciones (se verán en el Tema 5)
 - La clase `Random`.
 - La clase `StringJoiner`. 
 - La clase `Scanner`.
 - Contiene también interfaces y excepciones
 - ... consultar la documentación.

La clase `StringJoiner`

- Para crear una cadena con datos y delimitadores intermedio, inicial y final. El constructor recibe dichos delimitadores:

```
public StringJoiner(String delim,  intermedio  
                    String prefix,  inicial  
                    String suffix) ;  final
```

- Después para añadir los datos se usa el método:

```
public StringJoiner add(String s) ;
```

- Ejemplo

```
StringJoiner sj = new StringJoiner(" - ", "[", "]");  
sj.add("hola").add("que").add("tal") ;
```

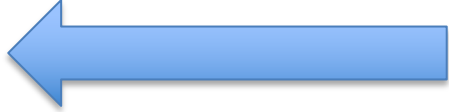
- entonces

```
sj.toString() ;
```

- es

```
"[hola - que - tal]"
```

El paquete `java.util`

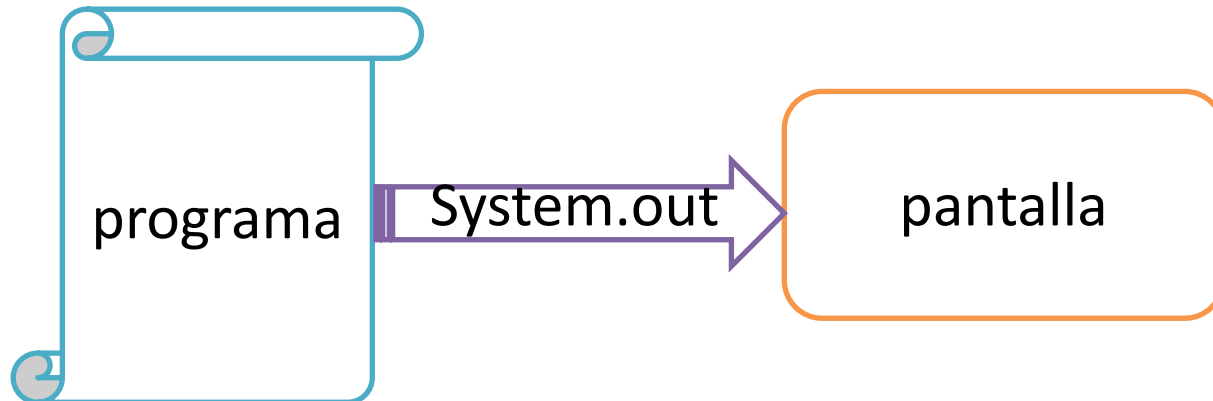
- Contiene clases de utilidad
 - Las colecciones (se verán en el Tema 5)
 - La clase **Random**.
 - La clase **StringJoiner**.
 - La clase genérica **Optional**.
 - La clase **Scanner**. 
 - Contiene también interfaces y excepciones
 - ... consultar la documentación.

La clase `Scanner`

- Ya hemos visto lo simple que es escribir datos por pantalla:

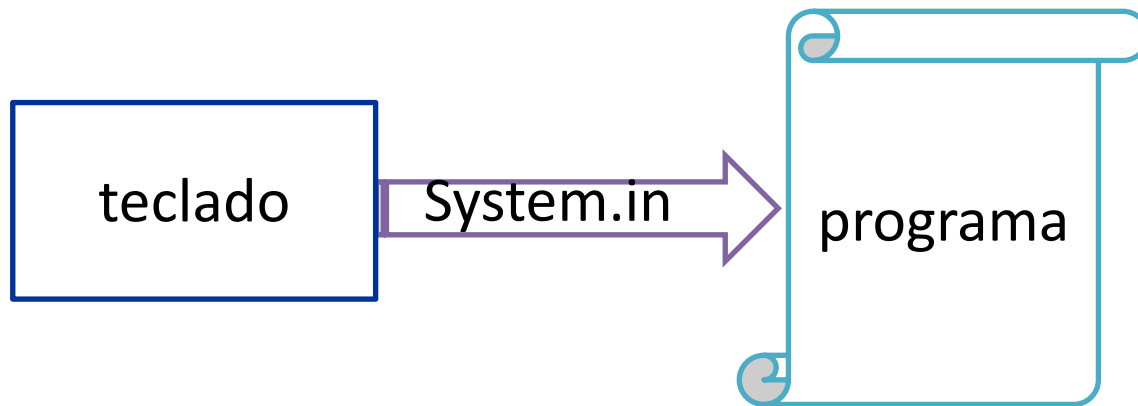
```
System.out.println  
System.out.print
```

- Con `System.out` accedemos a un objeto de la clase `System` conocido como el flujo de salida estándar (texto por la pantalla).



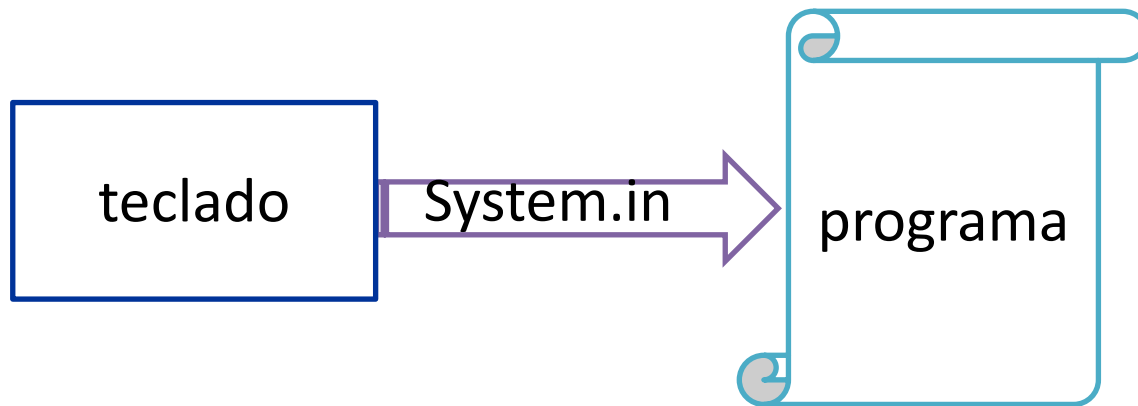
La clase `Scanner`

- De la misma forma , existe un `System.in` para el flujo de entrada estándar (texto desde el teclado).



La clase `Scanner`

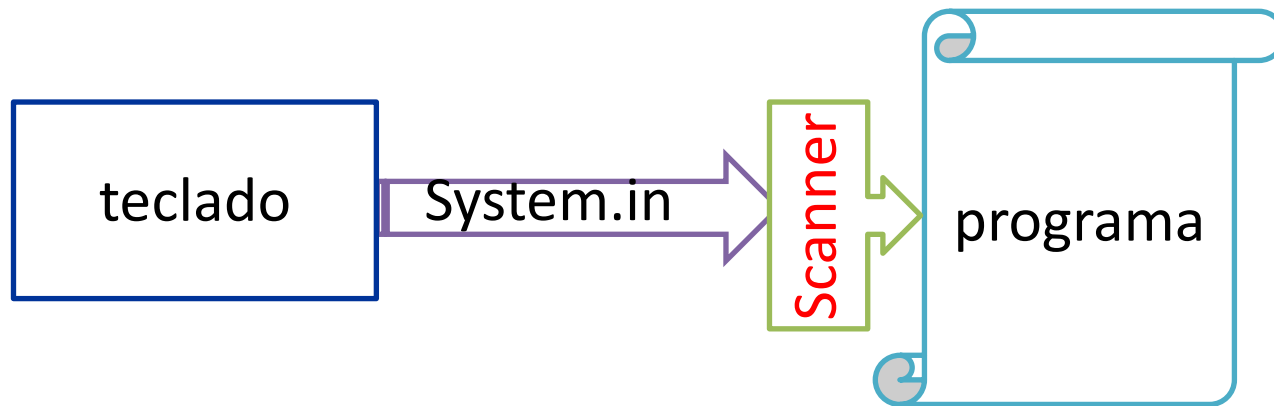
- Pero Java no fue diseñado para este tipo de entrada textual desde el teclado (modo consola).
- Por lo que `System.in` nunca ha sido simple de usar para este propósito (lectura de bytes).



La clase **Scanner**

- Afortunadamente, existe una forma fácil de leer datos desde la consola: objetos **Scanner**
- Al construir un objeto **Scanner**, se le pasa como argumento `System.in`:

```
Scanner teclado = new Scanner(System.in) ;
```



La clase `Scanner`

- La clase `Scanner` dispone de métodos de instancia para leer datos de diferentes tipos (por defecto los separadores son los espacios, tabuladores y nueva línea, aunque se pueden cambiar como ya veremos):
 - `next()` lee y devuelve el siguiente `String`
 - `nextLine()` lee y devuelve la siguiente línea como un `String`
 - `nextDouble()` lee y devuelve el siguiente `double`
 - `nextInt()` lee y devuelve el siguiente `int`
 - ...

todos primero saltan separadores

Ejemplo

```
import java.util.Scanner;
class EjScanner {
    static public void main(String[] args) {
        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();
        ...
    }
}
```

La clase **Scanner**

- Produce una excepción del tipo **InputMismatchException** si el dato a leer no es el esperado.
 - Por ejemplo si se utiliza `nextInt()` y lo siguiente no es un entero
- **InputMismatchException** hereda (indirectamente) de **RuntimeException**

La clase `Scanner`

- La clase `Scanner` también dispone de métodos para consultar si el siguiente dato disponible es de un determinado tipo:
 - `hasNextDouble()` devuelve `true` si el siguiente dato es un `double`
 - `hasNextInt()` devuelve `true` si el siguiente dato es un `int`
 - ...

Ejemplo

```
...  
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();  
...
```

- De esta forma evitamos que el sistema lance la excepción
- ¿Qué ocurre si la edad es negativa? ¿Cómo lo solucionamos?

La clase `Scanner`

- Por defecto `scanner` trata los números reales siguiendo la notación no inglesa, es decir utilizando la coma decimal en lugar del punto decimal. Por ejemplo: 4,5 en lugar de 4.5
- Para poder usar la notación inglesa debemos realizar la siguiente instrucción antes de la lectura (siendo `teclado` un objeto `scanner`):

```
teclado.useLocale(Locale.ENGLISH);
```

- Ahora podemos introducir 4.5 para leer un número real:

```
teclado.nextDouble();
```

- El valor por defecto es la coma como separador. Para volver a usar la coma como separador:

```
teclado.useLocale(Locale.FRENCH);
```

Delimitadores de 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:

<code>"[,;:]"</code>	<i>// Exactamente uno de entre ,;:</i>
<code>"[,;:]+"</code>	<i>// Uno o más de entre ,;:</i>
<code>"[^a-zA-Z0-9]"</code>	<i>// Cualquier carácter que no sea una letra o dígito</i>
<code>"[,;:] ?"</code>	<i>// Uno de entre ,;: seguido por un espacio opcional</i>
<code>"\\s*[, ; :]\\s*"</code>	<i>// Uno de entre ,;: entre múltiples [\\t\\n\\r\\f] opcionales</i>

La clase Scanner

- Existe una operación para “cerrar” el objeto Scanner, que es necesaria cuando ya no se va a utilizar más: `close()`

```
import java.util.Scanner;
class EjScanner {
    static public void main(String[] args) {
        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();
        ...
        teclado.close();
        //OJO. Ya no se puede leer nada de System.in
    }
}
```


La clase **Scanner**

- El cierre de un objeto **Scanner** se puede hacer automáticamente utilizando la instrucción **try** de la siguiente forma (tal y como se explicó en el tema 3 para cuando se tratan objetos “closeables”):

```
import java.util.Scanner;
class EjScanner {
    static public 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();
            ...
        } // OJO Se cierra la entrada System.in
    }
}
```

La clase **Scanner**

- Tanto si la ejecución termina con éxito como si se produce alguna excepción, el objeto **Scanner** será cerrado (más adelante insistiremos sobre esto al ver el paquete **java.io**)

```
import java.util.Scanner;
class EjScanner {
    static public 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();
            ...
        } catch (InputMismatchException e) {
            ...
        } // OJO Se cierra la entrada System.in
    }
}
```

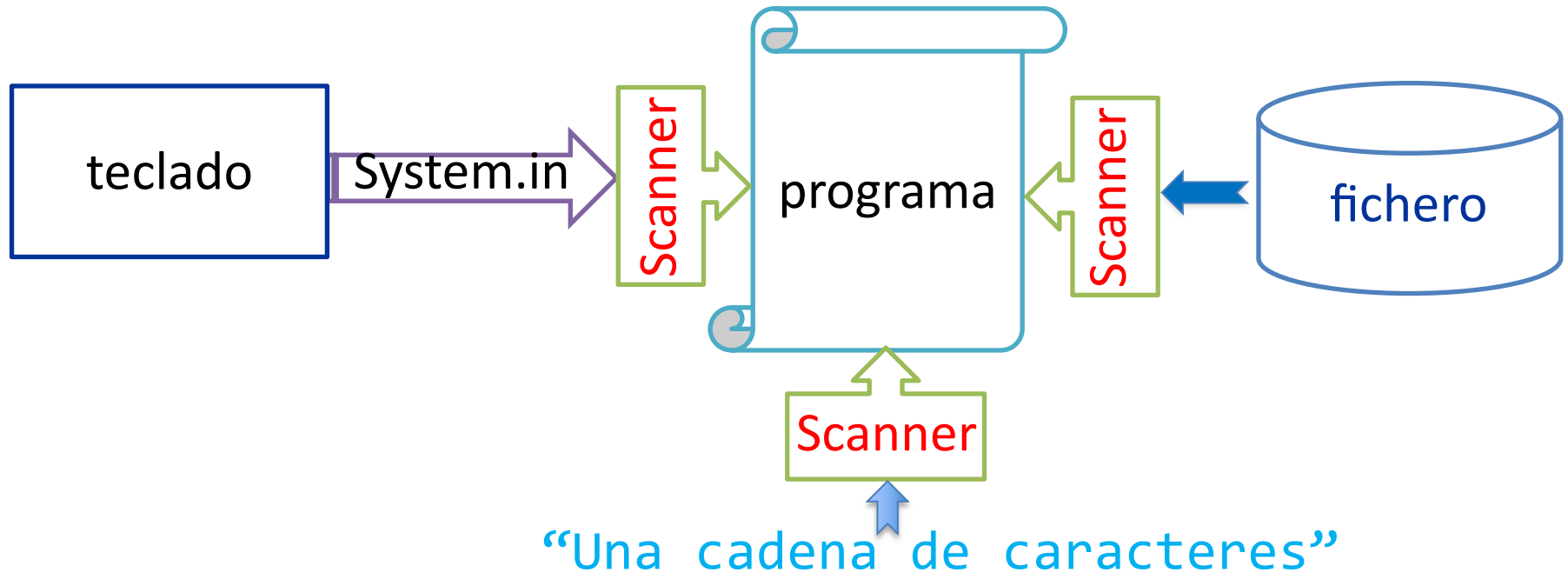
```

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
    }
}

```

La clase **Scanner**

- La clase **Scanner** no sólo sirve para leer de teclado.
- Se pueden construir objetos **Scanner** sobre objetos **String** y sobre objetos de otras clases de entrada de datos.



La clase **Scanner**

- Produce **NoSuchElementException** si no hay más elementos que obtener (fin de cadena, fin de fichero, ...)
- Produce **InputMismatchException** si el dato a obtener no es el esperado.
 - Por ejemplo si se utiliza `nextInt()` y lo siguiente no es un entero
- **InputMismatchException** hereda de **NoSuchElementException** y ésta a su vez de **RuntimeException**

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

```
// Pepe: 20  
// María: 30  
// Juan: 25  
// Lola: 22
```

```
try (Scanner sc = new Scanner("Pepe 7.2 María 8.5")) {  
    sc.useLocale(Locale.ENGLISH); // punto decimal  
        //Por defecto o 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);  
    System.out.println(nombre2 + ": " + nota2);  
} catch (InputMismatchException e) {  
    System.out.println("Error al extraer la nota");  
} catch (NoSuchElementException e) {  
    System.out.println("Error al extraer el dato");  
}
```

// Pepe: 7.2
// María: 8.5

La clase Scanner sobre un String

```
import java.util.Scanner;

public class Main {
    public static void main(String [] args) {
        try (Scanner sc = new Scanner("hola a ; todos. como-estas")) {
            // Separadores: espacio . , ; - una o mas veces (+)
            sc.useDelimiter("\\s*[. , ; -]+\\s*");
            while (sc.hasNext()) {
                System.out.println(sc.next());
            }
        }
    }
}
```

hola
a
todos
como
estas

Un analizador simple con la clase Scanner

"Juan García,23.Pedro González:15.Luisa López-19.Andrés Molina-22"

```
import java.util.Scanner;

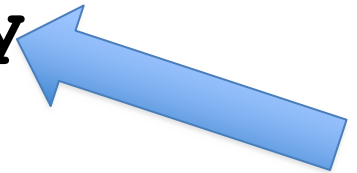
public class Main {
    public static void main(String [] args) {
        try (Scanner sc = new Scanner(args[0])) {
            sc.useDelimiter("[.]"); // Exactamente un punto
            while (sc.hasNext())
                tratarPersona(sc.next());
        }
    }

    private static void tratarPersona(String datosPersona) {
        try (Scanner scPersona = new Scanner(datosPersona)) {
            scPersona.useDelimiter("[,:-]"); // coma, dos puntos o guión
            String nombre = scPersona.next ();
            int edad = scPersona.nextInt();
            Persona persona = new Persona(nombre, edad);
            System.out.println(persona);
        }
    }
}
```

Persona(Juan García,23)
Persona(Pedro González,15)
Persona(Luisa López,19)
Persona(Andrés Molina,22)

Contenido

- Organización en paquetes
- Clases básicas: `java.lang`
- Clases útiles del paquete `java.util`
- Entrada/Salida (Ficheros)
 - los paquetes `java.io`, `java.nio` y `java.nio.file`



Entrada/Salida.

Los paquetes `java.io` y `java.nio`

- 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** se refiere a los datos que recibe el programa y la **salida** a los datos que transmite.
- Ya hemos visto la entrada de teclado y la salida a pantalla.
- Ahora con los paquetes **`java.io`**, **`java.nio`** y **`java.nio.file`** vamos a tratar algunos aspectos sencillos de la entrada/salida con **ficheros**.

Operaciones con ficheros

- **Apertura** – En esta operación se localiza e identifica un fichero existente, o bien se crea uno nuevo, para que se pueda operar con él. La apertura se puede realizar para leer o para escribir.
- **Escritura** – Para poder almacenar información en un fichero, una vez abierto en modo de escritura, hay que transferir la información organizada o segmentada 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 codificación de la información contenida en dicho fichero.
- **Cierre** – Cuando se va a dejar de utilizar un fichero se “cierra” la conexión entre el fichero y el programa. Esta operación se ocupa además de mantener la integridad del fichero, escribiendo previamente la información que se encuentre en algún buffer en espera de pasar al fichero.

La clase **Files** y la interfaz **Path**

- Se encuentran en el paquete `java.nio.file`
- Un **Path** representa un **camino abstracto** (independiente del S.O.) dentro de un sistema de ficheros.
 - Contiene información sobre el nombre y el camino de un fichero o de un directorio.
- Para construir un path se puede utilizar:
Path.of(String fichero)

```
Path p2 = Path.of("c:/users/juan/datos.txt");
```
- La clase **Files** utiliza estos objetos para operar con ficheros o directorios, créalos, borrarlos, saber si existen, obtener información, abrirlos para lectura, etc.

La clase **Files** . Métodos de clase:

- **Path createDirectory(Path)** crea un directorio
- **Path createDirectories(Path)** crea un directorio y los que hagan falta hasta llegar a él
- **Path createFile(Path)** crea un nuevo fichero
- **void delete(Path)**
- **boolean deleteIfExists(Path)**
- **boolean exists(Path)**
- **boolean isDirectory(Path)**
- **boolean isExecutable(Path)**
- **boolean isWritable(Path)**
- **List<String> readAllLines(Path)** crea una lista con todas las líneas del fichero
- Y muchos mas métodos interesantes

Lectura de fichero de texto.

`Files.readAllLines`

- 1) Crear un `Path` sobre un nombre de fichero

```
Path path= Path.of("datos.tex");
```

- 2) Leer todas las líneas del fichero en una lista con `Files.readAllLines` sobre el `Path` anterior

```
List<String> lineas = Files.readAllLines(path);
```

Este método se encarga de abrir el fichero, leer todas las líneas y cerrar el fichero.

Lee todo el fichero en memoria. Puede ser un problema si hay gran cantidad de datos.

Ejemplo: LeeConFiles

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class LeeConFiles {
    public static void main(String[] args) throws IOException {
        Path fichero = Path.of("personas.txt");
        for (String linea : Files.readAllLines(fichero))
            System.out.println(linea);
    }
}
```

Lee todo el fichero
en memoria

Lectura de un fichero con Scanner

- 1) Crear un **Path** sobre un nombre de fichero

```
Path path= Path.of("datos.tex");
```

- 2) Crear un **Scanner** sobre el **Path** anterior

```
Scanner sc = new Scanner(path);
```

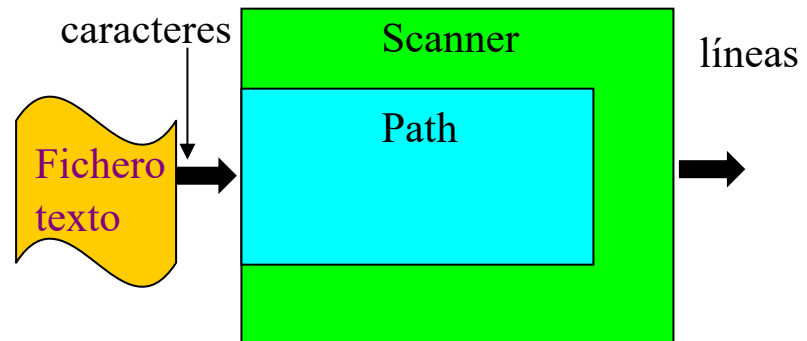
- 3) Leer líneas con **hasNextLine()** y **nextLine()** las veces que se necesite

```
while (sc.hasNextLine())  
    String linea = sc.nextLine();
```

- 4) Cerrar el **Scanner** (**Aquí es importante cerrarlo**)

```
sc.close();
```

Si se crea con **try** no
hay que cerrarlo



Ejemplo: LeeScanner

```
import java.io.IOException;
import java.nio.file.Path;
import java.util.Scanner;
public class LeeScanner {
    public static void main(String[] args) throws IOException {
        Path fichero = Path.of("personas.txt");
        try (Scanner sc = new Scanner(fichero)) {
            while (sc.hasNextLine())
                System.out.println(sc.nextLine());
        }
    }
}
```

Scanner
es Closeable

En memoria solo se tiene una
línea del fichero cada vez

Dos opciones para leer un fichero de texto.

1) **Files.readAllLines** Se lee todo el fichero en memoria colocando cada línea como un elemento en una lista de **String**.

- Si cabe en memoria es la forma más fácil de leer un fichero.

```
for (String linea : Files.readAllLines(fichero))  
    System.out.println(linea);
```

2) **Scanner** Se leen las líneas del fichero de una en una. Solo hay una línea en memoria en cada ciclo. Es **Closeable**.

- Si no cabe en memoria es la mejor forma de leer un fichero.

```
try (Scanner sc = new Scanner(fichero)) {  
    while (sc.hasNextLine())  
        System.out.println(sc.nextLine());  
}
```

La clase `PrintWriter`

- Permite escribir objetos y tipos básicos de Java sobre ficheros
- Constructor con el nombre de un fichero como argumento
`PrintWriter(String nombreFichero)`
- Métodos de instancia:

Para imprimir todos los tipos básicos y objetos

`print(...)` `println(...)` `printf(...)`

Escritura sobre un fichero de texto

- 1) Crear un **PrintWriter** sobre un nombre de fichero (se puede lanzar **FileNotFoundException**)

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

- 2) Escribir sobre el **PrintWriter**

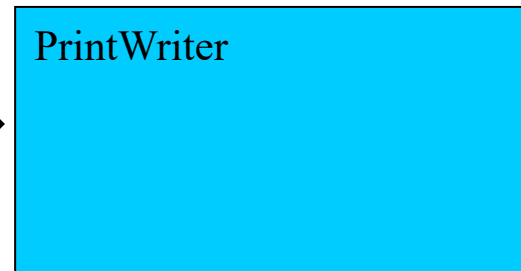
```
pw.println("Hola a todos");
```

- 3) Cerrar el **PrintWriter**

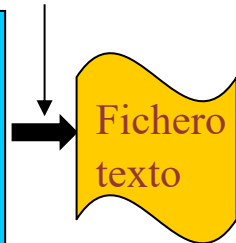
```
pw.close();
```

Si se crea en **try** no
hay que cerrarlo

"Hola a todos"



caracteres



Fichero
texto

Ejemplo

```
public void escribirFichero(String nombreFichero) throws FileNotFoundException {  
    // crear un fichero en el que almacenar varias líneas con palabras  
    PrintWriter pw = new PrintWriter(nombreFichero) ;  
    pw.println("amor roma mora ramo");  
    pw.println("rima mira");  
    pw.println("rail liar");  
    pw.close();  
}
```

Ejemplo

```
public void escribirFichero(String nombreFichero) throws FileNotFoundException {  
    // crear un fichero en el que almacenar varias líneas con palabras  
    try (PrintWriter pw = new PrintWriter(nombreFichero)) {  
        pw.println("amor roma mora ramo");  
        pw.println("rima mira");  
        pw.println("rail liar");  
    }  
}
```

**Mejor así (con try).
Cierre automático**

Ejemplo

```
public void escribirFichero(String nombreFichero) {  
    // crear un fichero en el que almacenar varias líneas con palabras  
    try (PrintWriter pw = new PrintWriter(nombreFichero)) {  
        pw.println("amor roma mora ramo");  
        pw.println("rima mira");  
        pw.println("rail liar");  
    } catch (FileNotFoundException e) {  
        System.err.println("ERROR: no se puede escribir en el fichero “  
                               + nombreFichero);  
    }  
}
```

(con try y capturando las excepciones)

PrintWriter y System.out

- Aunque ya sabemos que para mostrar datos por pantalla podemos utilizar de manera sencilla las operaciones que nos ofrece `System.out` (`print`, `println`, `printf`, ...), también podemos hacerlo con un objeto de la clase `PrintWriter`
- La clase `PrintWriter` tiene otro constructor que admite como parámetro `System.out`. Normalmente se utiliza con un segundo parámetro con el valor `true` (con objeto de que las salidas se realicen de inmediato (auto-flush)). Por ejemplo:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

¡OJO! En este caso no se debe cerrar el `PrintWriter`

- Esta opción es útil cuando creamos un método que recibe como parámetro un objeto de la clase `PrintWriter`, de manera que puede invocarse con objetos asociados a diferentes elementos (ficheros, pantalla, ...). [Esto se verá en las prácticas y es importante saberlo.](#)

PrintWriter y StringWriter

- A veces queremos disponer de la salida de una aplicación en un `String`. Para eso:
 - Usamos el constructor de `PrintWriter` con un argumento que es un `StringWriter`.

```
StringWriter st = new StringWriter();  
PrintWriter pw = new PrintWriter(st);
```

- Escribimos normalmente en el `PrintWriter` (y lo cerramos si es necesario).
- Después extraemos el `String` con

```
String salida = st.toString();
```

- Será muy útil cuando queramos escribir algo en un área de texto en una interfaz gráfica.

Ejemplo de StringWriter

```
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 void main(String[] args) {
        // Enviar información a un String a través de PrintWriter
        StringWriter salida = new StringWriter();
        try (pw = new PrintWriter(salida)) {
            imprimirDatos(pw);
            System.out.println(salida.toString());
        } catch (IOException e) {
            System.out.println("ERROR: no se puede escribir");
        }
    }
}
```