


Clases Básicas Predefinidas. Entrada/Salida

Contenido

- Organización en paquetes 
- Clases básicas del paquete **java.lang**
- Clases básicas del paquete **java.util**
- Entrada/Salida: clases básicas de los paquetes **java.io** 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 relacionadas desde un punto de vista lógico.
- Físicamente, un paquete se corresponde con un subdirectorio o carpeta.
- Las clases e interfaces de un paquete sólo se pueden acceder directamente por sus nombres desde otra clase o interfaz 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)

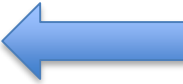
- La API es una biblioteca de paquetes que se suministra con la plataforma de desarrollo de Java (JDK).
- Estos paquetes contienen interfaces y clases diseñadas para facilitar la tarea de programación.
- En este tema veremos parte de los paquetes:

`java.lang`


`java.util`

`java.io` y `java.nio.file`

Contenido

- Organización en paquetes
- Clases básicas del paquete `java.lang` 
- Clases básicas del paquete `java.util`
- Entrada/Salida: clases básicas de los paquetes `java.io` 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**
 - 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;  
    }  
    @Override  
    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;  
    }  
    @Override  
    public boolean equals(Object o) {  
        return  
            (o instanceof Persona) &&  
            (edad == ((Persona) o).edad) &&  
            (nombre.equals(((Persona) o).nombre));  
    }  
}
```



Sin
Variable
auxiliar

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 `hashCode` para calcular el hashCode de cualquier clase:

```
Objects.hashCode (...);
```

`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.

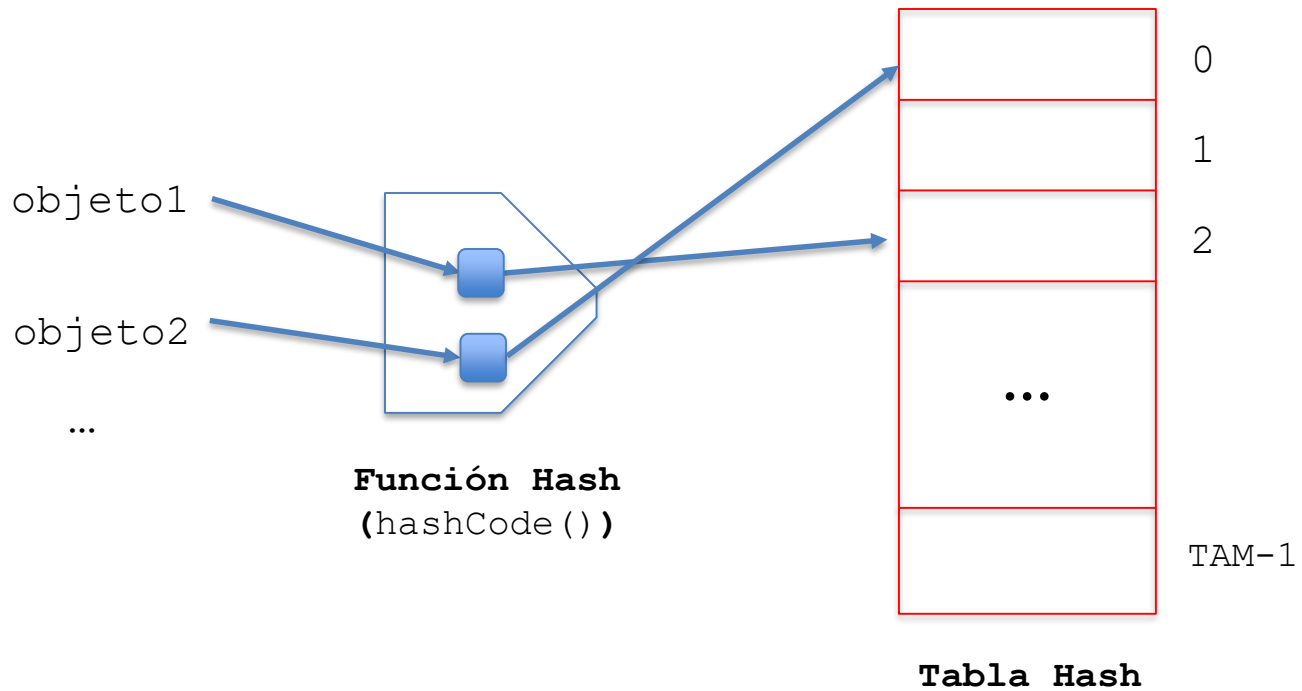
El que crea una clase es el responsable de mantener esta relación, redefiniendo adecuadamente los métodos:

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



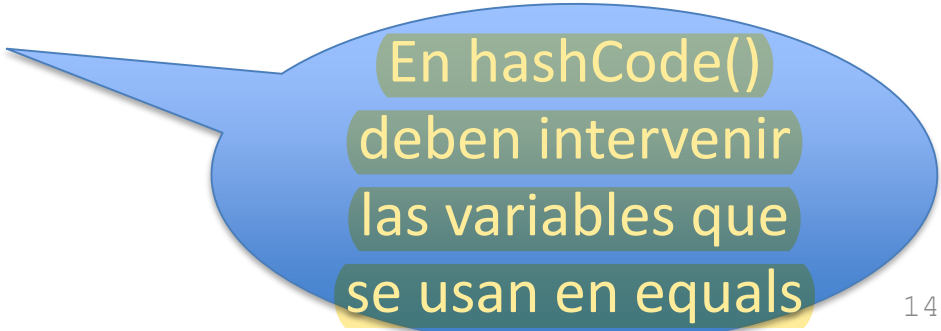
Fundamental para que posteriormente los objetos de esa clase puedan almacenarse correctamente en colecciones basadas en el uso de **Tablas Hash** (**Tema 6**)

`equals()` y `hashCode()`



equals () y hashCode ()

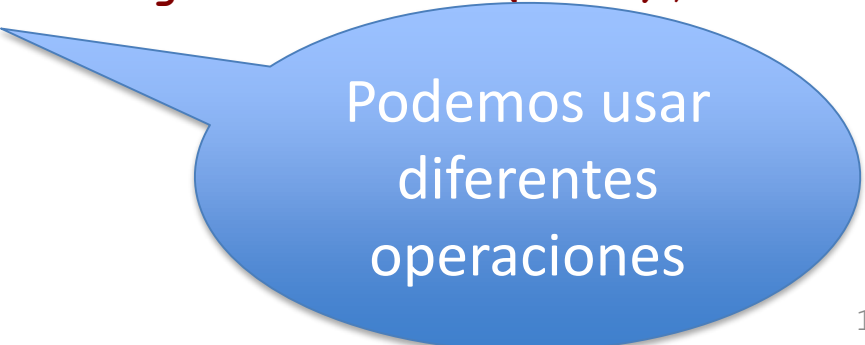
```
class Persona {  
    private String nombre;  
    private int edad;  
  
    public Persona(String n, int e) {  
        nombre = n;  
        edad    = e;  
    }  
    @Override  
    public boolean equals(Object o) {  
        return (o instanceof Persona) &&  
            (edad == ((Persona)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



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

equals () y hashCode ()

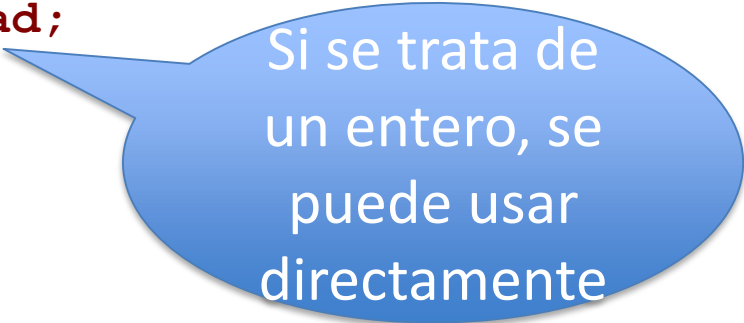
```
class Persona {  
    private String nombre;  
    private int edad;  
  
    public Persona(String n, int e) {  
        nombre = n;  
        edad    = e;  
    }  
    @Override  
    public boolean equals(Object o) {  
        return (o instanceof Persona) &&  
            (edad == ((Persona)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

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

equals () y hashCode ()

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

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


equals () y hashCode ()

```
class Persona {
    private String nombre;
    private int edad;

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

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


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**
 - Envoltorios de tipos básicos
 - ...
- Contiene interfaces:
 - **Comparable**
 - **Iterable**
 - ...
- Contiene también excepciones:
 - **ArrayIndexOutOfBoundsException**
 - **NullPointerException**
 - ...

La clase `System`

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

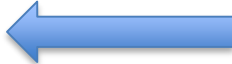
El paquete java.lang

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

La clase **Math**

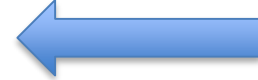
- Incorpora como *variables y métodos de clase* (estáticos) constantes y funciones matemáticas:
 - Constantes
 - `double E`, `double PI`
 - Métodos :
 - `double sin(double)`, `double cos(double)`, `double tan(double)`, `double asin(double)`, `double acos(double)`, `double atan(double)`, ...
 - `xxx abs(xxx)`, `xxx max(xxx,xxx)`, `xxx min(xxx,xxx)`, `double exp(double)`, `double pow(double, double)`, `double sqrt(double)`, `int round(float)`, ...
 - ...

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** 
 - Envoltorios de tipos básicos
 - ...
- Contiene interfaces:
 - **Comparable**
 - **Iterable**
 - ...
- Contiene también excepciones:
 - **ArrayIndexOutOfBoundsException**
 - **NullPointerException**
 - ...

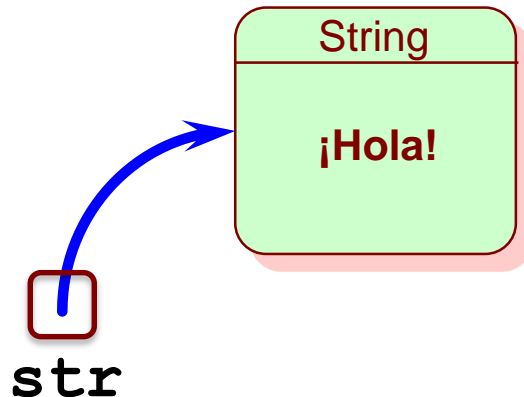
Cadenas de caracteres

- Las cadenas de caracteres literales constantes se representan en Java como secuencias de caracteres Unicode encerradas entre comillas dobles.
- Para manipular cadenas de caracteres utilizaremos dos clases incluidas en **java.lang**:
 - **String** - para cadenas inmutables
 - **StringBuilder** - para cadenas modificables



La clase **String**

- Ya vimos algunas características en el tema 2
- Cada objeto de tipo **String** almacena 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**

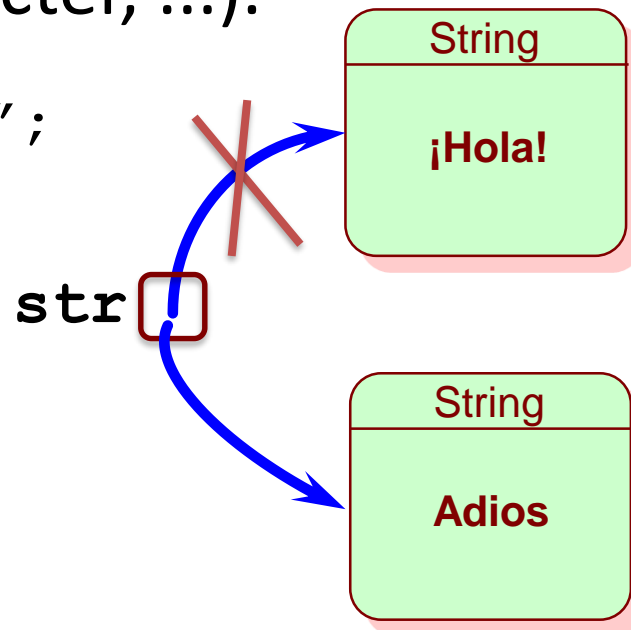
- Ya vimos algunas características en el tema 2
- A una **variable** de tipo **String** se le puede **asignar cadenas distintas** durante su existencia.
- Pero **una cadena de caracteres** almacenada en un objeto de tipo **String** **NO puede modificarse** (crecer, cambiar un carácter, ...).

```
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)`

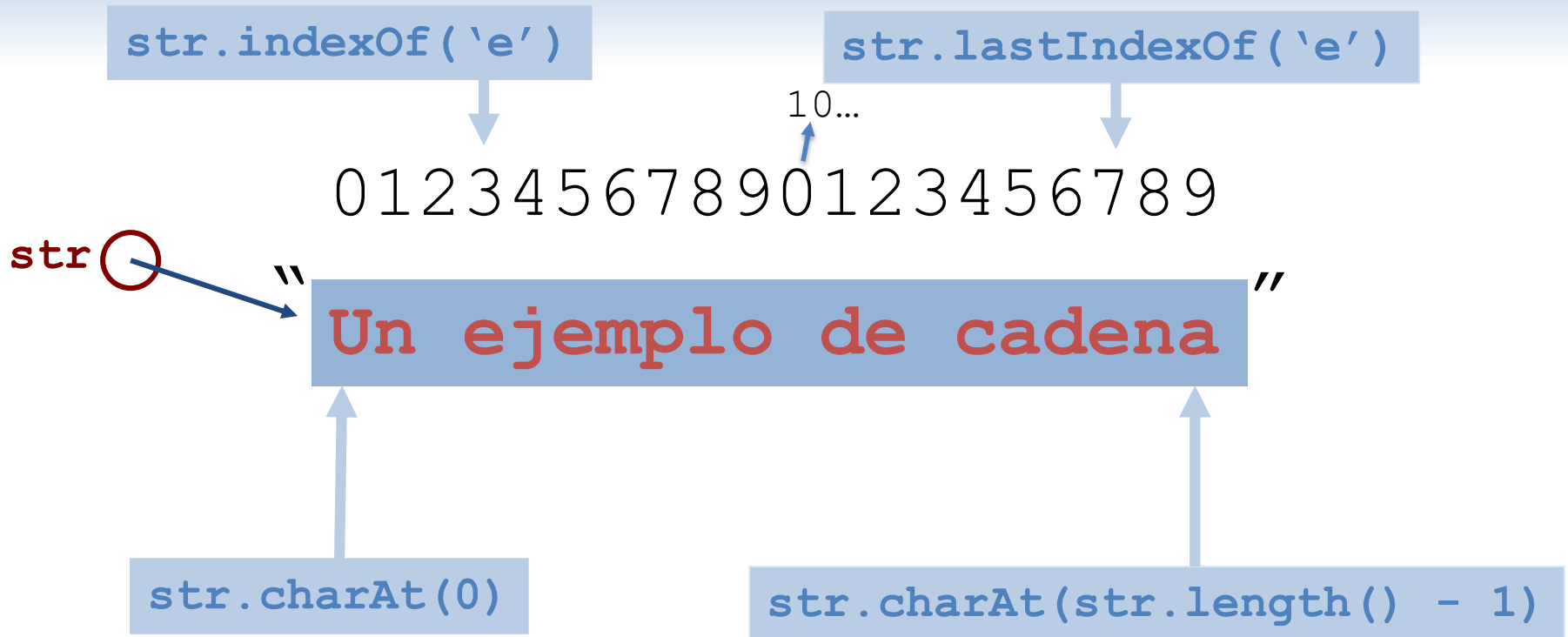
...

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

`IndexOutOfBoundsException`

`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:

`IndexOutOfBoundsException`

`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 <code>toString()</code> .	<code>"%s"</code>	<code>"%20s"</code>
– d	para números sin decimales.	<code>"%d"</code>	<code>"%7d"</code>
– f	para números con decimales.	<code>"%f"</code>	<code>"%9.2f"</code>
– b	para booleanos	<code>"%b"</code>	<code>"%5b"</code>
– c	para caracteres.	<code>"%c"</code>	<code>"%4c"</code>

- 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`

- Métodos que producen nuevos objetos `String`:

`String [] split(String delimitadores)`

Permite extraer subcadenas de una cadena en base a unos delimitadores (expresiones regulares)

Métodos de la clase `String`

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` será `["", "juan", "garcia", "17", "", "", "carpintero"]`

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

```
String str = ":juan garcia;17..,carpintero";  
String [] items = str.split("[ , . ; : ]+");
```

`items` será `["", "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

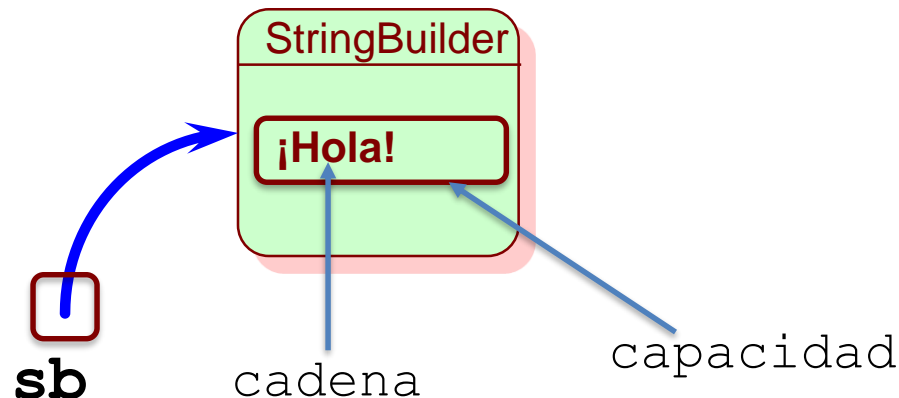


La clase `StringBuilder`

- Al igual que un objeto de tipo `String`, cada objeto de tipo `StringBuilder` **almacena 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();
```

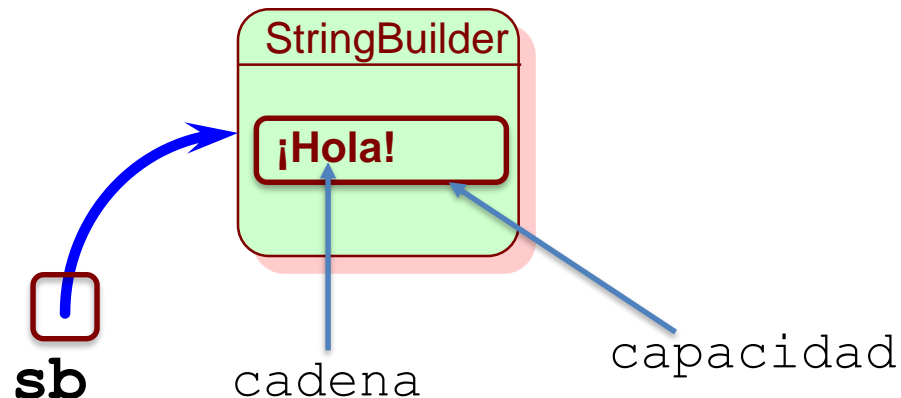
```
// cadena vacía "" y capacidad inicial para 16 caracteres
```

```
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
```



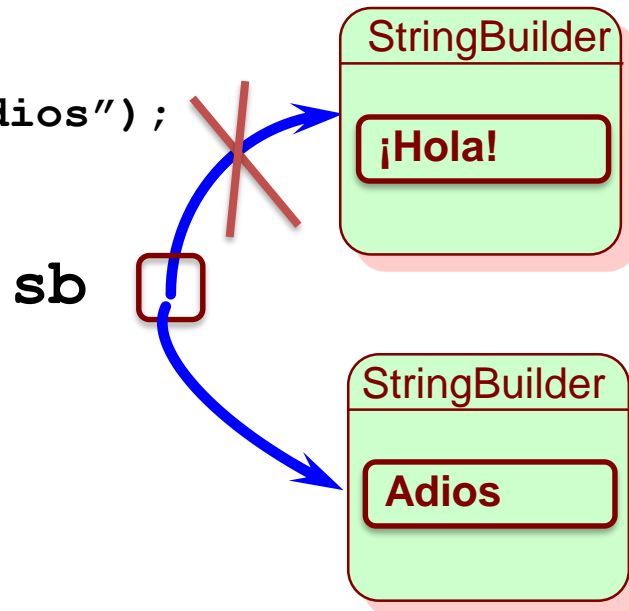
La clase `StringBuilder`

- Al igual que a una variable de tipo `String`, a una variable de tipo `StringBuilder` se le puede asignar cadenas distintas durante su existencia.

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

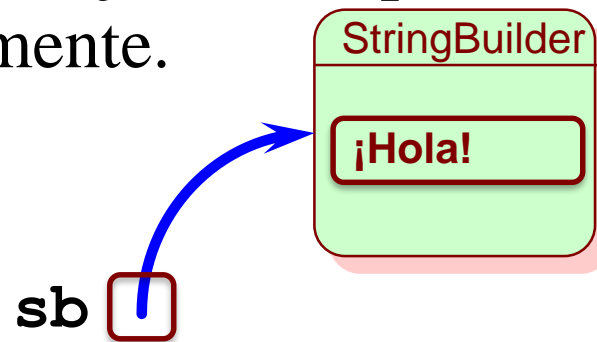
```
...
```

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



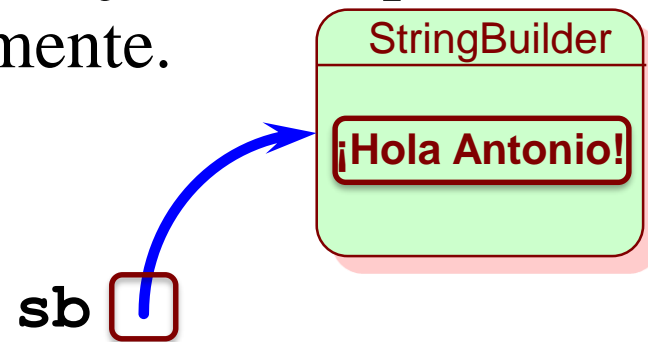
La clase `StringBuilder`

- Pero así no es como se utilizan los objetos de tipo `StringBuilder` normalmente.
- 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.
- Y así es como se utilizan los objetos de tipo `StringBuilder` normalmente.



La clase `StringBuilder`

- Pero así no es como se utilizan los objetos de tipo `StringBuilder` normalmente.
- 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.
- Y así es como se utilizan los objetos de tipo `StringBuilder` normalmente.



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)`
(no tiene las versiones para char)

(pero dada una variable `c` de tipo `char`, podemos poner `c+""` como primer parámetro en esas operaciones)

...

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

`IndexOutOfBoundsException`

`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:

```
IndexOutOfBoundsException
```

```
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:

`IndexOutOfBoundsException`

`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()
...
```

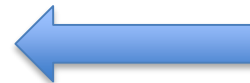
```
StringBuilder sb = new StringBuilder("esto es un ejemplo");
...
sb.setCharAt(0, 'E');           // sb será "Esto es un ejemplo"
sb.replace(8, 10, "otro");      // sb será "Esto es otro ejemplo"
```

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

```
IndexOutOfBoundsException
StringIndexOutOfBoundsException
```

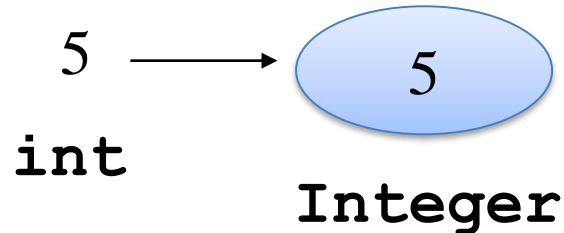
El paquete java.lang

- Siempre está incluido en cualquier aplicación, no es necesario importarlo explícitamente.
- Contiene las clases básicas del sistema:
 - **Object**
 - **System**
 - **Math**
 - **String, StringBuilder**
 - 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 listas, como vimos en el Tema 2 (ArrayList <Integer>))
- 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

- Constructores: crean envoltorios a partir de los datos numéricos o cadenas de caracteres:

```
Integer oi = new Integer(34); // o new Integer("34");
```

```
Double od = new Double(34.56); // o new Double("34.56");
```

“Deprecated, for removal: This API element is subject to removal in a future version”

- Métodos de clase para crear envoltorios a partir de los datos numéricos o cadenas de caracteres:

```
static Xxx valueOf(xxx n)
```

```
Integer oi = Integer.valueOf(34);
```

```
Double od = Double.valueOf(34.56);
```

```
static Xxx valueOf(String s)
```

```
Integer oi = Integer.valueOf("34");
```

```
Double od = Double.valueOf("34.56");
```

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

Los envoltorios numéricos

- Métodos de instancia para extraer el dato numérico del envoltorio:

```
xxxx xxxxValue()  
    int    i    = oi.intValue() ;  
    double d    = od.doubleValue() ;
```

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

```
static xxxx parseXxxx(String s)  
    int        i = Integer.parseInt("234") ;  
    double     d = Double.parseDouble("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 n1, xxxx n2)
    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 n)
    int r1 = Integer.hashCode(45);
    int r2 = Double.hashCode(34.25);
    int r3 = Long.hashCode(45);
```

El envoltorio Boolean

- Constructores: crean envoltorios a partir de valores lógicos o cadenas de caracteres:

```
Boolean ob = new Boolean(false); // o new Boolean("false");
```

“Deprecated, for removal: This API element is subject to removal in a future version”

- Métodos de clase para crear envoltorios a partir de valores lógicos o cadenas de caracteres:

```
static Boolean valueOf(boolean b)
```

```
Boolean ob = Boolean.valueOf(false);
```

```
static Boolean valueOf(String s)
```

```
Boolean ob = Boolean.valueOf("false");
```

- Si la cadena no representa un valor lógico (se ignora mayúsculas o minúsculas) no produce error, sino que lo toma como **false**

El envoltorio Boolean

- Método de instancia para extraer el valor lógico del envoltorio:

```
boolean booleanValue()
```

```
boolean b = ob.booleanValue();
```

- Método de clase para crear un valor lógico a partir de cadenas de caracteres:

```
static boolean parseBoolean(String s)
```

```
boolean b = Boolean.parseBoolean("true");
```

- Si la cadena no representa un valor lógico (se ignora mayúsculas o minúsculas) no produce error, sino que lo toma como **false**

El envoltorio Boolean

- Método de clase para comparar datos lógicos:

```
static int compare(boolean b1, boolean b2)  
    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 b)  
    int r = Boolean.hashCode(true);
```

El envoltorio Character

- Constructor: crea un envoltorio a partir de un carácter:

```
Character oc = new Character('a');
```

“Deprecated, for removal: This API element is subject to removal in a future version”

- Método de clase para crear un envoltorio a partir de un carácter:

```
static Character valueOf(char c)
```

```
Character oc = Character.valueOf('a');
```

El envoltorio Character

- Método de instancia para extraer el dato carácter del envoltorio:

```
char charValue()  
  
char c = oc.charValue();
```

- Métodos de clase para comprobar tipos de caracteres:

```
static boolean isDigit(char c)  
static boolean isLetter(char c)  
static boolean isLowerCase(char c)  
static boolean isUpperCase(char c)  
static boolean isSpaceChar(char c)  
  
boolean b = Character.isLowerCase('g');
```

- Métodos de clase para obtener caracteres mayúsculas o minúsculas:

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

El envoltorio Character

- Método de clase para comparar caracteres:

```
static int compare(char c1, char c2)
    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 c)
    int r = Character.hashCode('a');
```

Envolver y desenvolver automáticamente (*boxing/unboxing*)

- El compilador realiza de forma automática la *conversión* de tipos básicos a objetos y viceversa.

```
List<Double> lista = new ArrayList<>();
```


```
...
```

```
ENVUELVE lista.add(45.5);
```

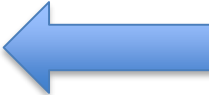
```
...
```

```
DESENVUELVE double d = 5.2 + lista.get(j);
```

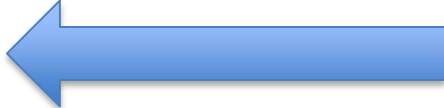
El paquete java.lang

- Siempre está incluido en cualquier aplicación, no es necesario importarlo explícitamente.
- Contiene las clases básicas del sistema:
 - `Object`
 - `System`
 - `Math`
 - `String`, `StringBuilder`
 - Envoltorios de tipos básicos
 - ...
- Contiene interfaces:  **Tema 6**
 - `Comparable`
 - `Iterable`
 - ...
- Contiene también excepciones:
 - `ArrayIndexOutOfBoundsException`
 - `NullPointerException`
 - ...

Contenido

- Organización en paquetes
- Clases básicas del paquete `java.lang`
- Clases básicas del paquete `java.util` 
- Entrada/Salida: clases básicas de los paquetes `java.io` y `java.nio.file`

El paquete `java.util`

- Contiene clases de utilidad
 - **Las colecciones** (se verán en el Tema 6)
 - La clase **Random**. 
 - La clase **StringJoiner**.
 - La clase genérica **Optional**.
 - 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)  
...
```

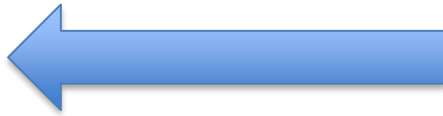
- Consultar la documentación para información adicional.

La clase Random

- Ejemplo:
programa para calcular el valor medio de un millón de números reales (entre 0 y 1 (sin incluirlo)) generados aleatoriamente.

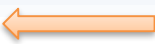


```
import java.util.Random;
public class TestAleatorio {
    public static void main(String[] args) {
        Random rnd = new Random();
        double sum = 0.0;
        for (int i = 0; i < 1000000; i++) {
            sum += rnd.nextDouble();
        }
        System.out.println("media = " + sum / 1000000.0);
    }
}
```

El paquete `java.util`

- Contiene clases de utilidad
 - Las colecciones (se verán en el Tema 6)
 - La clase **Random**.
 - La clase **StringJoiner**. 
 - La clase genérica **Optional**.
 - La clase **Scanner**.
 - Contiene también interfaces y excepciones
 - ... consultar la documentación.

La clase `StringJoiner`

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

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

- Después para añadir los datos se usa el método:

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

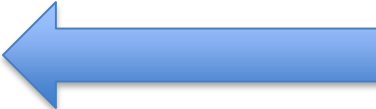
- Ejemplo

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

CON `sj.toString()` obtenemos "[hola - que - tal]"

Muy útil para diseñar el método `toString()` de las clases

El paquete `java.util`

- Contiene clases de utilidad
 - Las colecciones (se verán en el Tema 6)
 - La clase **Random**.
 - La clase **StringJoiner**.
 - La clase genérica **Optional**. 
 - La clase **Scanner**.
 - Contiene también interfaces y excepciones
 - ... consultar la documentación.

La clase genérica `Optional`

- En el Tema 6 hablaremos de clases genéricas (aunque ya hemos visto algo al manejar listas en el Tema 2).
- Un objeto `Optional<T>` puede contener o no un dato de la clase `T`.

```
Optional<String> o1 = Optional.of("hola");
```

```
Optional<String> o2 = Optional.empty();
```

- Métodos de instancia:

```
boolean isPresent() // indica si hay dato o no
```

```
T get() // devuelve el dato almacenado o lanza  
        // NoSuchElementException si no hay nada
```

```
T orElse(T v) // devuelve el dato almacenado o  
               // devuelve v si no hay nada
```

- La clase tiene correctamente definidos `equals` y `hashCode`

La clase genérica Optional

(ejemplo 1)

```
public static Persona buscar(List<Persona> datos, String nombre) {  
    int i = 0;  
    while ((i < datos.size()) && (!nombre.equals(datos.get(i).nombre())))  
        i++;  
    return (i < datos.size()) ? datos.get(i) : null;  
}
```

```
public static void main(String [] args) {  
    List<Persona> datos = new ArrayList<>();  
    ...  
    Persona p = buscar(datos, "Pepe");  
    if (p != null) {  
        System.out.println(p);  
    } ...  
}
```

La clase genérica **Optional**

(ejemplo 1)

```
public static Optional<Persona> buscar(List<Persona> datos, String nombre) {  
    int i = 0;  
    while ((i < datos.size()) && (!nombre.equals(datos.get(i).nombre())))  
        i++;  
    return (i < datos.size()) ? Optional.of(datos.get(i)) : Optional.empty();  
}
```

```
public static void main(String [] args) {  
    List<Persona> datos = new ArrayList<>() ;  
    ...  
    Optional<Persona> op = buscar(datos, "Pepe");  
    if (op.isPresent()) {  
        System.out.println(op.get());  
    } ...  
}
```

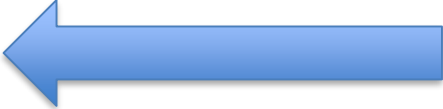

La clase genérica Optional

(ejemplo 2)

```
public class Urna {  
    ...  
    public ColorBola extraeBola() {  
        if (totalBolas() == 0) {  
            throw new RuntimeException("...");  
        }  
        ...  
    }  
}
```

```
public class Urna {  
    ...  
    public Optional<ColorBola> extraeBola() {  
        Optional<ColorBola> res;  
        if (totalBolas() == 0) {  
            res = Optional.empty();  
        } else {  
            ...  
        }  
        ...  
    }  
}
```

El paquete `java.util`

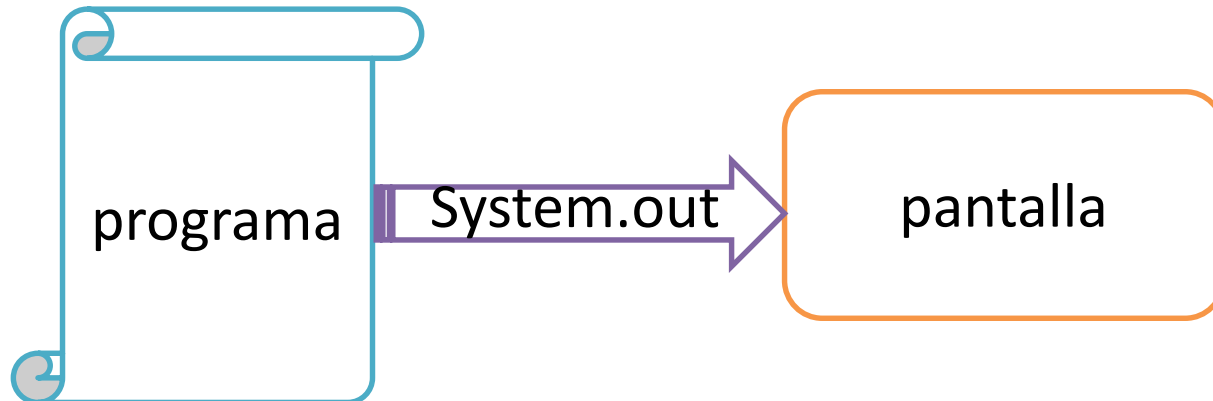
- Contiene clases de utilidad
 - Las colecciones (se verán en el Tema 6)
 - La clase **Random**.
 - La clase **StringJoiner**.
 - La clase genérica **Optional**.
 - La clase **Scanner**. 
 - Contiene también interfaces y excepciones
 - ... consultar la documentación.

La clase **Scanner**

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

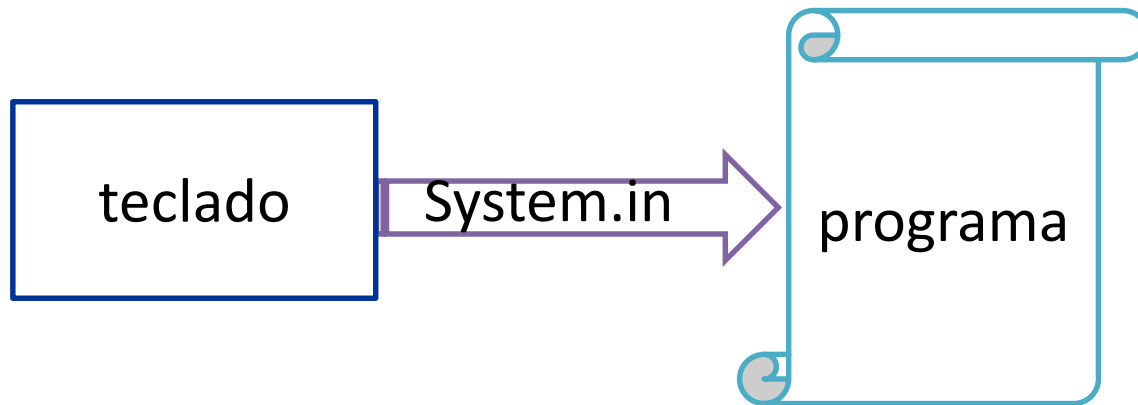
```
System.out.println  
System.out.print
```

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



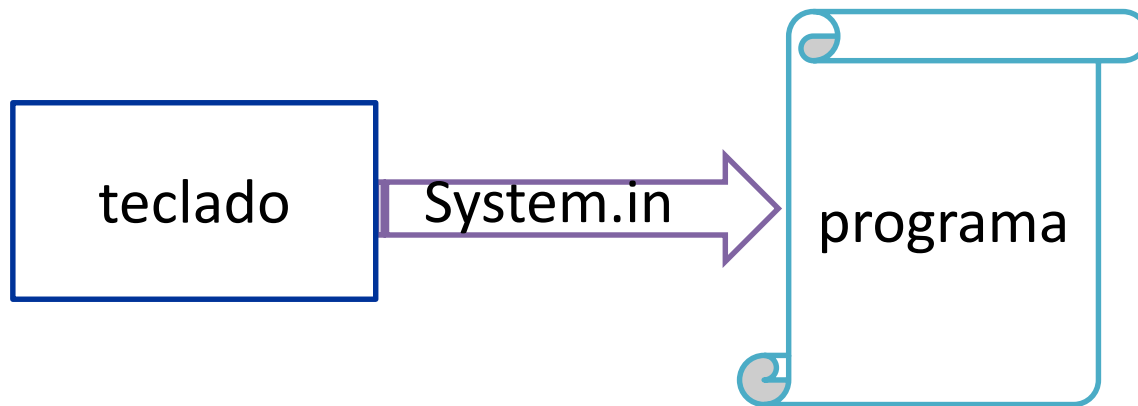
La clase Scanner

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



La clase `Scanner`

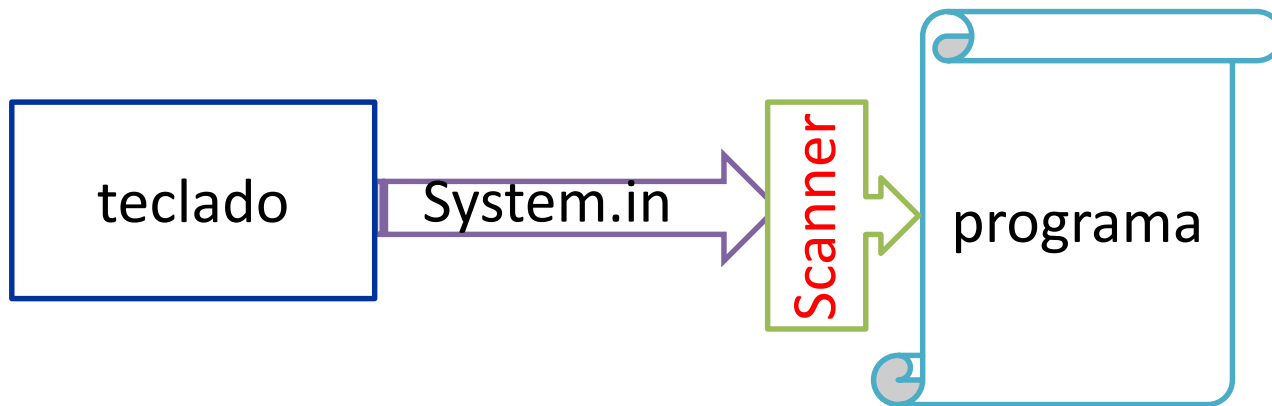
- Pero Java no fue diseñado para este tipo de entrada textual desde el teclado (modo consola).
- Por lo que `System.in` nunca ha sido simple de usar para este propósito (lectura de bytes).



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 obtener datos de diferentes tipos (por defecto los separadores (antes y después del dato a obtener) son los espacios, tabuladores y salto de línea, aunque se pueden cambiar como ya veremos):
 - `next()` obtiene y devuelve el siguiente `String`
 - `nextLine()` obtiene y devuelve la siguiente línea como un `String`
 - `nextDouble()` obtiene y devuelve el siguiente `double`
 - `nextInt()` obtiene y devuelve el siguiente `int`
 - ...

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 obtener 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

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 (siendo `teclado` un objeto `Scanner`):

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

- Ahora podemos obtener 4.5 como número real:

```
teclado.nextDouble()
```

La clase `Scanner`

- Por defecto los separadores son los espacios, tabuladores y salto de línea, pero se pueden establecer otros así (siendo `teclado` un objeto `Scanner`):

```
teclado.useDelimiter(delimitadores)
```

- **Los `delimitadores`: Expresiones Regulares**

- Ejemplos

- “[, : .]” Exactamente uno de entre , : . y espacio
- “[, : .]+” Uno o más de entre , : . y espacio

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();
    }
}
```

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

La clase **Scanner**

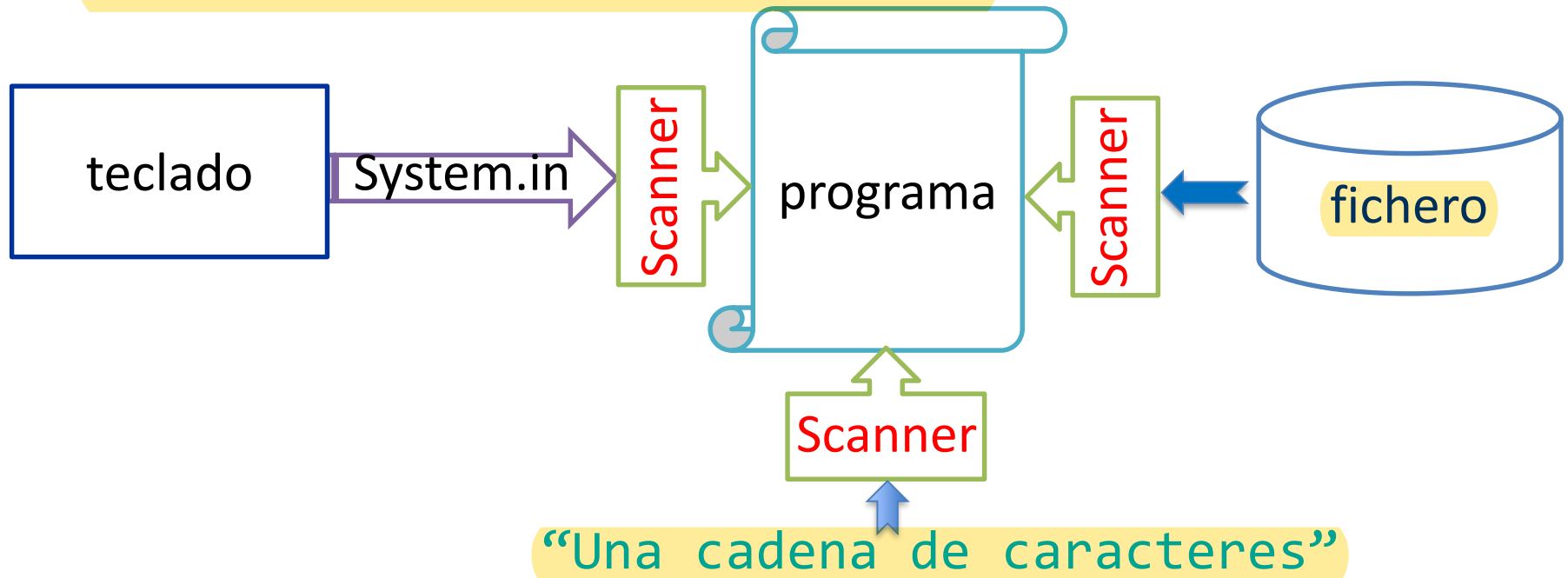
- Tanto si la ejecución termina con éxito como si se produce alguna excepción, el objeto **Scanner** será cerrado (más adelante insistiremos sobre esto al ver los paquetes **java.io** y **java.nio.file**)

```
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) {
            ...
        }
    }
}
```


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**

La clase **Scanner** sobre un **String**

(Ejemplo 1)

```
import java.util.Scanner;
```

```
public class Ejemplo1 {  
    public static void descomponer(String cadena) {  
        try (Scanner sc = new Scanner(cadena)) {  
            // Separadores: espacio . , ; - una o mas veces (+)  
            sc.useDelimiter("[.,;-]+");  
            while (sc.hasNext()) {  
                System.out.println(sc.next());  
            }  
        }  
    }  
}
```

```
...  
}
```

hola
a
todos
como
estais

```
Ejemplo1.descomponer("hola a ; todos. como-estais");
```

Alternativa: usar método `split` de `String`

(Ejemplo 1)

```
public class Ejemplo1 {  
    public static void descomponer(String cadena) {  
        // Separadores: espacio . , ; - una o mas veces (+)  
        String [] subcadenas = cadena.split("[.,;-]+");  
        for (String s : subcadenas) {  
            System.out.println(s);  
        }  
    }  
    ...  
}
```

hola
a
todos
como
estais

```
Ejemplo1.descomponer("hola a ; todos. como-estais");
```

La clase **Scanner** sobre un **String**

(Ejemplo 2)

Ejemplo2.analizar("Juan García,23.Pedro González:15.Luisa López-19.Andrés Molina-22");

```
import java.util.Scanner;
```

```
public class Ejemplo2 {  
    public static void analizar(String cadena) {  
        try (Scanner sc = new Scanner(cadena)) {  
            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);  
            ...  
        }  
    }  
}
```

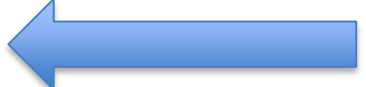
Alternativa: usar método `split` de `String`

(Ejemplo 2)

`Ejemplo2.analizar("Juan García,23.Pedro González:15.Luisa López-19.Andrés Molina-22");`

```
public class Ejemplo2 {
    public static void analizar(String cadena) {
        String [] subcadenas = cadena.split("[.]"); // Exactamente un punto
        for (String s : subcadenas) {
            tratarPersona(s);
        }
    }
    private static void tratarPersona(String datosPersona) {
        String [] subcadenas = datosPersona.split("[,:-]"); // coma, dos puntos o guión
        String nombre = subcadenas[0];
        int edad = Integer.parseInt(subcadenas[1]);
        Persona persona = new Persona(nombre, edad);
        ...
    }
}
```

Contenido

- Organización en paquetes
- Clases básicas del paquete `java.lang`
- Clases básicas del paquete `java.util`
- **Entrada/Salida:** clases básicas de los paquetes `java.io` y `java.nio.file` 

Entrada/Salida. Los paquetes `java.io` y `java.nio.file`

- 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`** 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 interfaz **Path**

- Se encuentra 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 (carpeta).
- Para construir un objeto “path” se puede utilizar:
Path.of(String nombre)

```
Path fichero = Path.of("c:/users/juan/datos.txt");
```

```
Path fichero = Path.of("datos.txt");
```

La clase **Files**

- Se encuentra en el paquete `java.nio.file`
- La clase **Files** utiliza los objetos “path” para operar con ficheros o directorios: crearlos, borrarlos, saber si existen, obtener información, abrirlos para lectura, etc.
 - `Path createDirectory(Path)`
 - `Path createFile(Path)`
 - `void delete(Path)`
 - `boolean deleteIfExists(Path)`
 - `boolean exists(Path)`
 - `boolean isDirectory(Path)`
 - `boolean isExecutable(Path)`
 - `boolean isWritable(Path)`
 - `BufferedReader newBufferedReader(Path)` 
 - ...

La clase `BufferedReader`

- Proporciona eficiencia a la hora de leer
- Utiliza un buffer intermedio
- Lanza **`IOException`** si hay problemas con alguna de las operaciones
- Métodos de instancia

```
String readLine() // lectura de una línea  
                // devuelve null si no hay
```

```
void close()      // cierre del flujo
```

```
...
```

Lectura de fichero (con Path y Files)

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

```
Path fichero = Path.of("datos.txt");
```

- 2) Crear un **BufferedReader** usando la clase **Files**

```
BufferedReader br = Files.newBufferedReader(fichero);
```

- Lanza una excepción **IOException** (del paquete **java.io**) o **subclase de ella** si el fichero no se puede abrir

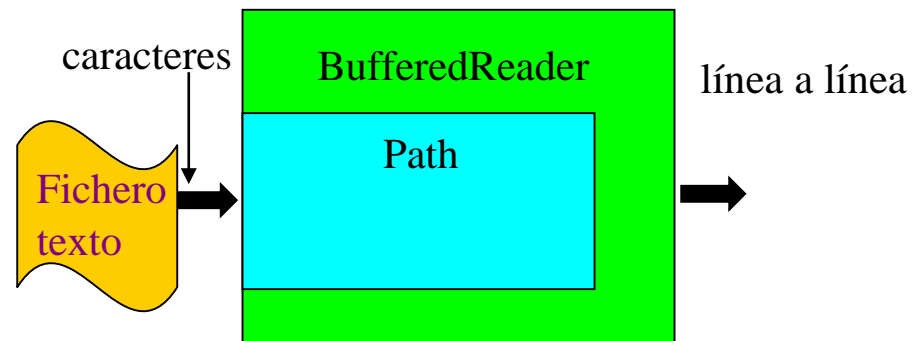
- 3) Leer línea a línea con **readLine()**

```
String linea = br.readLine();
```

- 4) Cerrar el **BufferedReader**

```
br.close();
```

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



Ejemplo

```
// leer el fichero y mostrar las diferentes líneas por pantalla
public void mostrarLineasFichero (String nombreFichero) throws IOException {
    Path fichero = Path.of(nombreFichero);
    BufferedReader br = Files.newBufferedReader(fichero);
    String linea = br.readLine();
    while (linea != null) {
        System.out.println(linea);
        linea = br.readLine();
    }
    sc.close();
}
```

Ejemplo

```
// leer el fichero y mostrar las diferentes líneas por pantalla
public void mostrarLineasFichero (String nombreFichero) throws IOException {
    Path fichero = Path.of(nombreFichero);
    try (BufferedReader br = Files.newBufferedReader(fichero)) {
        String linea = br.readLine();
        while (linea != null) {
            System.out.println(linea);
            linea = br.readLine();
        }
    }
}
```

Mejor así (con try).
Cierre automático

Ejemplo

```
// leer el fichero y mostrar las diferentes líneas por pantalla
public void mostrarLineasFichero (String nombreFichero) {
    Path fichero = Path.of(nombreFichero);
    try (BufferedReader br = Files.newBufferedReader(fichero)) {
        String linea = br.readLine();
        while (linea != null) {
            System.out.println(linea);
            linea = br.readLine();
        }
    } catch (IOException e) {
        System.err.println("ERROR: no se puede leer del fichero "
                           + nombreFichero);
    }
}
```

(con try y capturando las excepciones)

Lectura de fichero (con Path y Scanner)

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

```
Path fichero = Path.of("datos.txt");
```

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

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

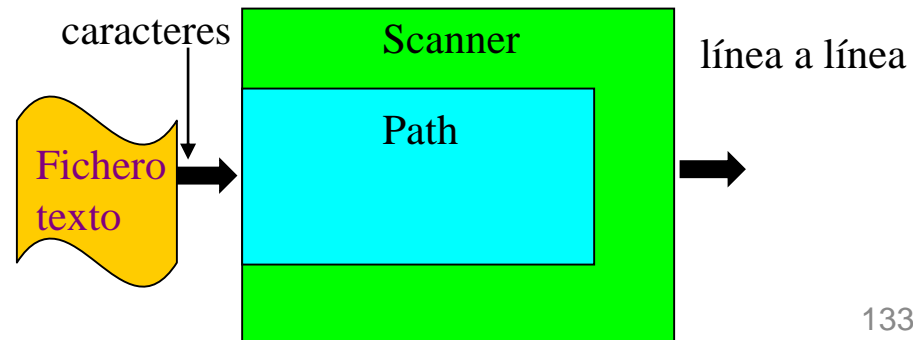
- Lanza una excepción **IOException** (del paquete `java.io`) o **subclase de ella** si el fichero no se puede abrir.

- 3) Leer línea a línea con **hasNextLine()** y **nextLine()**

- 4) Cerrar el **Scanner**

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

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



Ejemplo

```
// leer el fichero y mostrar las diferentes líneas por pantalla
public void mostrarLineasFichero (String nombreFichero) throws IOException {
    Path fichero = Path.of(nombreFichero);
    Scanner sc = new Scanner(fichero);
    while (sc.hasNextLine()) {
        System.out.println(sc.nextLine());
    }
    sc.close();
}
```

Ejemplo

```
// leer el fichero y mostrar las diferentes líneas por pantalla
public void mostrarLineasFichero (String nombreFichero) throws IOException {
    Path fichero = Path.of(nombreFichero);
    try (Scanner sc = new Scanner(fichero)) {
        while (sc.hasNextLine()) {
            System.out.println(sc.nextLine());
        }
    }
}
```

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

Ejemplo

```
// leer el fichero y mostrar las diferentes líneas por pantalla
public void mostrarLineasFichero (String nombreFichero) {
    Path fichero = Path.of(nombreFichero);
    try (Scanner sc = new Scanner(fichero)) {
        while (sc.hasNextLine()) {
            System.out.println(sc.nextLine());
        }
    } catch (IOException e) {
        System.err.println("ERROR: no se puede leer del fichero "
                           + nombreFichero);
    }
}
```

(con try y capturando las excepciones)

La clase `PrintWriter`

- Se encuentra en el paquete `java.io`
- 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 (puede lanzar **FileNotFoundException**)

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

- 2) Escribir sobre el **PrintWriter**

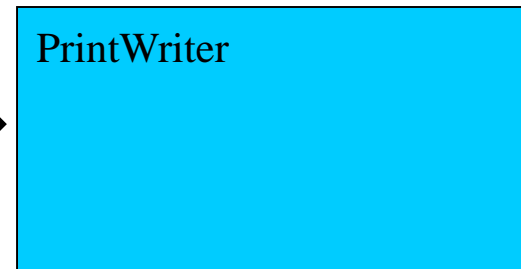
```
pw.println("Hola a todos");
```

- 3) Cerrar el **PrintWriter**

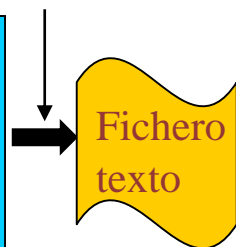
```
pw.close();
```

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

"Hola a todos"



caracteres



Fichero
texto

Ejemplo

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

Ejemplo

```
// crear un fichero en el que almacenar varias líneas con palabras
public void escribirFichero(String nombreFichero) throws FileNotFoundException {
    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

```
// crear un fichero en el que almacenar varias líneas con palabras
public void escribirFichero(String nombreFichero) {
    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 FileWriter

- Usando el constructor de `PrintWriter` como hemos hecho anteriormente (con el nombre de un fichero como argumento), si el fichero ya existía, se destruye su información y se escribe sobre un fichero vacío
- Si queremos añadir más información a un fichero ya existente, debemos usar un constructor de `PrintWriter` que admite un objeto de la clase `FileWriter` (especificando en el constructor de ésta el valor `true` como segundo parámetro). Por ejemplo:

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

- Si en el constructor de `FileWriter` se especifica el valor `false` como segundo parámetro, de nuevo se destruye la información del fichero existente y se escribe sobre uno vacío.
- La clase `FileWriter` también se encuentra en el paquete `java.io`

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** (se encuentra también en el paquete **java.io**).

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

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

- Escribimos normalmente en el **PrintWriter** (y lo cerramos si es necesario).
- Extraemos el **String** con

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

- Será muy útil cuando queramos escribir algo en un área de texto en una interfaz gráfica (**GUIs – Tema 5**).