Clases Básicas Predefinidas

Contenido

- Organización en paquetes
- Clases básicas: **`java.lang`**
- Clases básicas del paquete **`java.util`**
- Entrada/Salida. El paquete **`java.io`**

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, con una protección de acceso, que delimita un ámbito para el uso de nombres.
- La plataforma Java incorpora unos paquetes predefinidos para facilitar determinadas acciones.
- Se pueden definir paquetes nuevos

Creación de paquetes

- Para definir un paquete hay que encabezar cada fichero que componga el paquete con la declaración

```
package <nombre>;
```

- Cuando no aparece esta declaración, se considera que las clases e interfaces de los ficheros pertenecen a un paquete anónimo.

Uso de paquetes

- Desde fuera de un paquete sólo se puede acceder a clases e interfaces **`public`** (exceptuando el acceso a clases herederas en el caso de declaraciones **`protected`**).
- Para acceder desde otro paquete a una clase o interfaz se puede:
  - Utilizar el nombre calificado con el nombre del paquete

    ```
    gráfico.Rectángulo r;
    ```
  - importarla al comienzo del fichero y usar su nombre simple

    ```
    import gráfico.Rectángulo;
    ```
  - importar el paquete completo al comienzo del fichero y usar los nombres simples de todas las clases e interfaces del paquete

    ```
    import gráfico.*;
    ```
- El sistema de ejecución de Java importa de forma automática el paquete anónimo, **`java.lang`** y el paquete actual.

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`**

El paquete java.lang

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

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 definición de clase no extiende a otra, entonces extiende a **`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:
  - **`String toString()`**
  - **`boolean equals(Object o)`**
  - **`int hashCode()`**
  - **`Object clone()`**
  - **`Class getClass()`**
  - **`void finalize()`**
  - ... consultar la documentación.

El método `equals()`

- Compara dos objetos de la misma clase.
- Por defecto realiza una comparación por **==**.
- Este método se puede redefinir en cualquier clase para comparar objetos de esa clase.
- Todas las clases del sistema tienen redefinido este método.

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

equals() y hashCode()

- El método **`hashCode()`** devuelve un **`int`** para cada objeto de la clase.
- Hay una relación que debe mantenerse entre equals() y hashCode();

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

- **Todas las clases del API de Java verifican esa relación**.
- Para los tipos básicos existen clases que nos permitirán calcular su hashcode.

int	Integer
short	Short
long	Long
double	Double
char	Character
bool	Boolean

Ejemplos

```
Double.hashCode(34.56)
Integer.hashCode(-98)
Boolean.hashCode(true)
```

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

- boolean equals(Object)
- Int hashCode()

Ejemplo1 de **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) && (p.nombre.equals(nombre));
    }
    public int hashCode() {
        return nombre.hashCode() + Integer.hashCode(edad);
    }
}
```

Ejemplo2 de `equals y hashCode`

- Si en la definición de equals, se utilizan algunas variables de instancias, esas mismas deben usarse para calcular el hashCode. (Cuidado con los String)

```
class Persona {
    private String nombre;
    private int edad;
    public Persona(String n, int e) {
        nombre = n;
        edad = e;
    }
    public boolean equals(Object obj) {
        boolean res = obj instanceof Persona;
        Persona per = res ? (Persona)obj: null;
        return res && edad == per.edad &&
                    nombre.equalsIgnoreCase(per.nombre);
    }
    public int hashCode() {
        return nombre.toLowerCase().hashCode() +
                                    Integer.hashCode(edad);
    }
}
```

La clase `System`

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

    provoca la ejecución inmediata de los **`finalize()`** pendientes
  - …
- Consultar documentación para más información.

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(double),…
double random(),
```

    - …
  - –Consultar la documentación para información adicional.

Ej.: `System.out.println(Math.sqrt(34));`

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, por razones de eficiencia, 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 cosas en el tema 2
- Cada objeto alberga una cadena de caracteres.
- Los objetos de esta clase se pueden inicializar...
  - de la forma normal:

```
String str = new String("¡Hola!");
```

  - de la forma simplificada:

```
String str = "¡Hola!";
```

- A una variable **`String`** se le puede asignar cadenas distintas durante su existencia.
- Pero una cadena de caracteres almacenada en una variable **`String`** NO puede modificarse (crecer, cambiar un carácter, ...).

Métodos de la clase `String`

- Métodos de consulta:

```
int length()
char charAt(int pos)
int indexOf/lastIndexOf(char car)
int indexOf/lastIndexOf(String str)
...
```

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

  `IndexOutOfBoundsException`

Métodos de la clase **`String`**

- Métodos que producen nuevos objetos **`String`**:

```
String substring(int posini, int posfin+1)
String substring(int posini)
String replace(String str1, String str2)
String concat(String s)   // también con +
String toUpperCase()
String toLowerCase()
static String format(String formato,...)
...
```

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

  `IndexOutOfBoundsException`

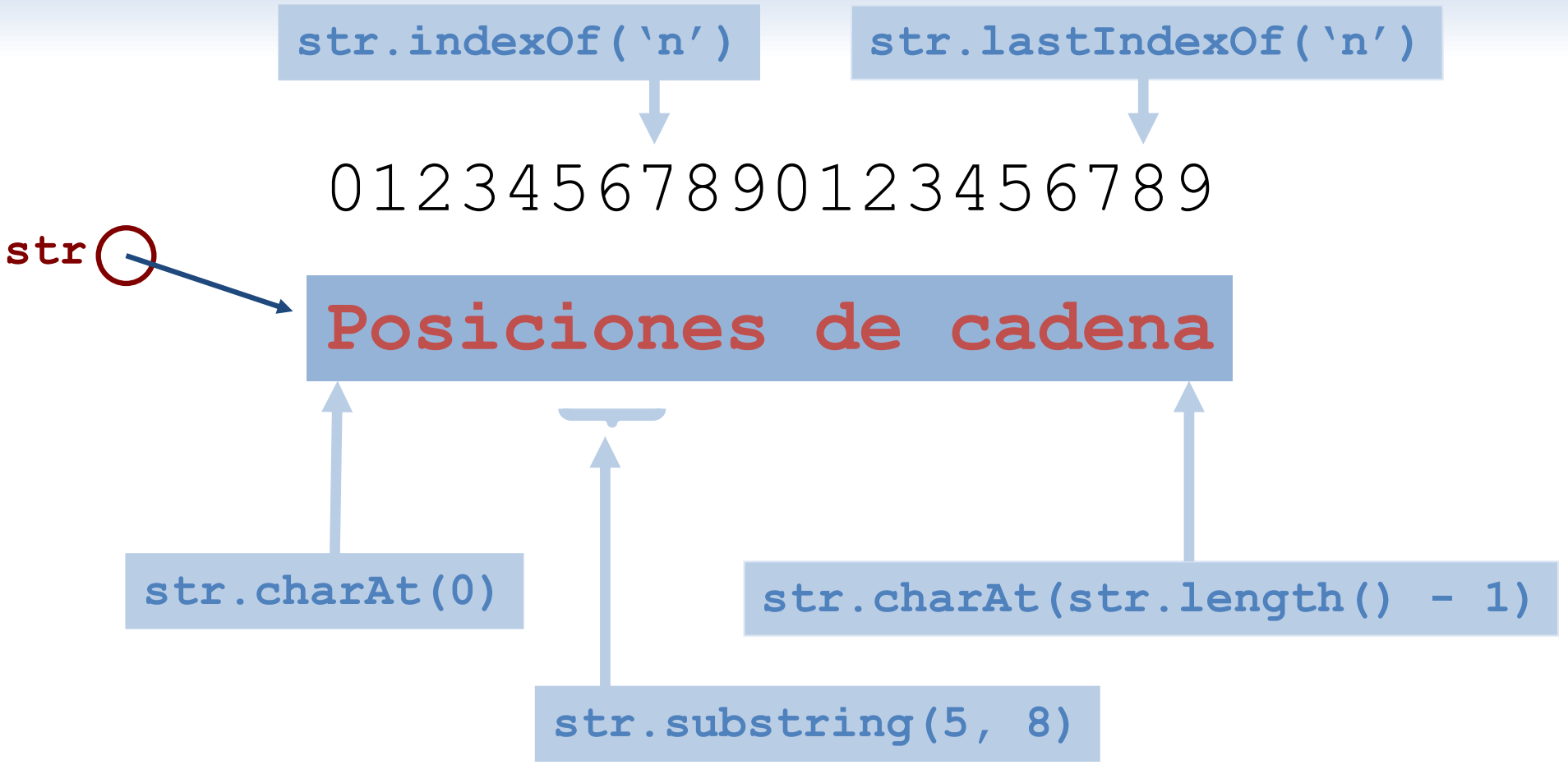
Métodos de la clase `String`

- Comparación:

```
boolean equals(String str)
boolean equalsIgnoreCase(String str)
int compareTo(String str)
int compareToIgnoreCase(String str)
```

- ¡ojo!
  - cadena1 == cadena2 compara referencias

Posiciones de una cadena



```
public class NombreFichero {
        private String camino;
        private char separadorCamino, separadorExtensión;

        public NombreFichero(String str, char sep, char ext) {
                camino = str;
                separadorCamino = sep;
                separadorExtensión = ext;
        }
        public String extensión() {
                int pto = camino.lastIndexOf(separadorExtensión);
                return camino.substring(pto + 1);
        }
        public String nombre() {
                int pto = camino.lastIndexOf(separadorExtensión);
                int sep = camino.lastIndexOf(separadorCamino);
                return camino.substring(sep + 1, pto);
        }
        public String directorio() {
                …
        }
                ...
}
```

El método estático `format`

- A partir de JDK 1.5.
- 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);
```

- 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 usa el símbolo – para ajustar a la izquierda "%-20s"
- Se pueden producir las excepciones:
  - MissingFormatArgumentException
  - IllegalFormatConversionException
  - UnknownFormatConversionException
  - ...

El método estático **format**

- Las clases **PrintStream** y **PrintWriter** incluyen el método **printf(String formato,...)**

```
class EjPf {
        static public void main(String[] args) {
          String s = String.format("El objeto %20s con %d", new A(65), 78);
          System.out.println(s);
          System.out.printf(
            "Cadena %40s\nEntero %15d\nFlotante %8.2f\nLógico %b\n",
            "Esto es una cadena", 34, 457.2345678, 3 == 3);
         }
}
class A {
   int a;
   public A(int s) {
         a = s;
   }
   public String toString() {
         return "A[“ + a + “]";
   }

}
```

```
El objeto                 A[65] con 78
Cadena                         Esto es una cadena
Entero                 34
Flotante    457,23
Lógico true
```

El método Split

Permite extraer datos de una cadena según unos delimitadores:

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

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

```
String [] items1 = "hola.a to;dos".split("[ ,.;:]");
items1->{"hola","a","to","dos"}
```

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

```
String [] items2 =
 "juan garcia;17..,carpintero".split("[ ;.,]+");
items2->{"juan","garcia","17","carpintero"}
```

La clase **StringBuilder**

- Cada objeto alberga una cadena de caracteres.

- Los objetos de esta clase se inicializan de cualquiera de las formas siguientes:

```
StringBuilder strB = new StringBuilder(10);
StringBuilder strB2 = new StringBuilder("ala");
```

- Las cadenas de los objetos **StringBuilder** se pueden ampliar, reducir y modificar mediante mensajes.

- Cuando la capacidad establecida se excede, se aumenta automáticamente.

Métodos de la clase `StringBuilder`

- Métodos de consulta:

```
int length()
int capacity()
char charAt(int pos)
int indexOf/lastIndexOf(String str)
...
```

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

```
IndexOutOfBoundsException
```

Métodos de la clase **`StringBuilder`**

- Métodos para construir objetos **`String`**:

```
String substring(int posini, int posfin+1)
String substring(int posini)
String toString()
...
```

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

`IndexOutOfBoundsException`

Métodos de la clase **`StringBuilder`**

- Métodos para modificar objetos **`StringBuilder`**:

```
StringBuilder append(String str)
StringBuilder insert(int pos, String str)
StringBuilder setCharAt(int pos, char car)
StringBuilder replace(int pos1,int pos2+1,
                      String str)
StringBuilder reverse()
...
```

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

  `IndexOutOfBoundsException`

Métodos de la clase **`StringBuilder`**

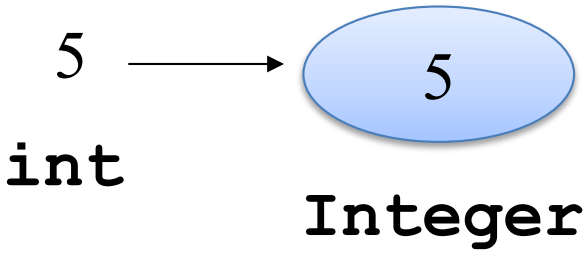
- La clase **`StringBuilder`** no tiene definidos los métodos para realizar comparaciones que tiene la **`String`**.
- Pero se puede usar el método **`toString()`** para obtener un **`String`** a partir de un **`StringBuilder`** y poder usarlo para comparar.

Ejemplo

```
public class StringDemo {
      public static void main(String[] args) {
            String cadena = “Aarón es Nombre“;
            int long = cadena.length();
            StringBuilder réplica = new StringBuilder(long);
            char c;
            for (int i = 0; i < long; i++) {
                  c = cadena.charAt(i);
                  if (c == ‘A’) {
                        c = ‘V’;
                  } else if (c == ‘N’) {
                        c = ‘H’;
                  }
                  réplica.append(c)
            }
            System.out.println(réplica);
      }
}
```

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
- 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);
Double  od = new Double("34.56");
```

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

```
xxxx xxxxValue()
```

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

- Se le han añadido método de clases para definir algoritmos sobre los tipos básicos

Los envoltorios numéricos. Métodos de clase

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

xxxx parseXxxx(String)

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

- Métodos de clase para comparar datos básico:

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
```

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

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

int hashCode(xxxx v)

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

El envoltorio **Boolean**

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

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

- 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:

`boolean parseBoolean(String)`

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

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

`Boolean valueOf(String)`

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

- Si el dato introducido no es lógico no produce error, sino que lo toma como **`false`**
- **Método de clase para calcular el hashCode**

```
int r4 = Boolean.hashCode(true);
```

El envoltorio `Character`

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

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

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

```
char charValue()
```

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

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

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

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

- Métodos de clase para comparar caracteres:

```
int compare(char, char)
```

- Métodos de clase para convertir caracteres:

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

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

Envolver y desenvolver automáticamente (*boxing/unboxing)*

- El compilador realiza de forma automática la *conversión* de tipos básicos a objetos y viceversa.
- No es posible enviar un mensaje a valores de tipo básico.

```
                Double [] lista = new Double[TAM];
                ...
   ENVUELVE     lista[i] = 45.5;
                ...
DESENVUELVE     double d = 5.2 + lista[j];
```

El paquete `java.util`

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

La clase **Random**

- Los objetos representan variables aleatorias de distinta naturaleza:

```
Random r = new Random();
```

- Permite generar números aleatorios de diversas formas:

```
float nextFloat()
double nextDouble()
int nextInt(int n)    // 0 <= res < n
```

…

- Consultar la documentación para información adicional.

La clase `StringJoiner`

- Para crear una cadena con datos y delimitadores intermedio, inicial y final.

```
public StringJoiner(String delim,
                    String prefix,
                    String suffix);
```

- Para añadir elemento 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]"

La clase genérica `Optional`

- En el tema de colecciones hablaremos de clases genéricas.
- Un objeto Optional<T> puede contener un dato de una clase T o no.

```
Optional<String> os1 = Optional.of("hola");
Optional<String> os2 = Optional.empty();
```

- Métodos de instancia:

```
boolean isPresent() // indica si hay dato o no
T get()    // noSuchElementoException si no hay nada
T orElse(T o) // Devuelve el dato y si no hay o
```

- La clase tiene correctamente definido equals y hashCode

La clase genérica `Optional`

```
public static Optional<Persona> buscar(Persona datos[], String nombre) {
    int i = 0;
    while ((i < datos.length) && (!nombre.equals(datos[i].getName()))) {
        ++i;
    }
    Optional<Persona> r;
    if (i < datos.length) {
        r = Optional.of(datos[i]);
    } else {
        r = Optional.empty();
    }
    return r;
}
```

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

Alternativa a lanzar una excepción

En la clase **`Recta`** del ejemplo **`prRecta`**

```
public Optional<Punto> interseccionCon(Recta r) {
  if (paralelaA(r)) {
   return Optional.empty();
  } else {
  ...
}
```

En la clase **`Urna`** del proyecto **`prUrna`**

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

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)
- 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.

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 para leer datos de diferentes tipos (por defecto los separadores son los espacios, tabuladores y nueva línea):
  - `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`
  - …

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 **NoSuchElementException** si no hay más elementos que leer
- Produce **InputMismatchException** si el dato a leer no es el esperado.
  - Por ejemplo si se utiliza `nextInt()` y lo siguiente no es un entero

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?

La clase `Scanner`

- Por defecto los separadores son los espacios, tabuladores y nueva línea, pero se pueden establecer otros:
  - `useDelimiter(String` delimitadores`)`

- 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, lo cual es necesario cuando ya no se vaya 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 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) {
            ...
        }
    }
}
```

Si se cierra un scanner sobre System.in, ya no se podrá leer mas desde System.in

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** sobre un **String**

```
import java.io.IOException;
import java.util.Scanner;

public class Main {
    public static void main(String [] args) {
        try (Scanner sc = new Scanner(args[0])) {
        // Separadores: espacio . , ; -    una o mas veces (+)
            sc.useDelimiter("[ .,;-]+");
            while (sc.hasNext()) {
                String cad = sc.next();
                System.out.println(cad);
            }
        }
    }
}
```

"hola a ; todos. como-estas"

```
hola
a
todos
como
estas
```

Un analizador simple con la clase **Scanner**

```
import java.util.Scanner;

public class Main {
    public static void main(String [] args) {
        String datos =
            "Juan García,23.Pedro González:15.Luisa López-19.Andrés Molina-22";
        try (Scanner sc = new Scanner(datos)) {
            sc.useDelimiter("[.]");  // Exactamente un punto
            while (sc.hasNext()) {
                String datoPersona= sc.next();
                try (Scanner scPersona = new Scanner(datoPersona)) {
                    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);
                }
            }
        }
    }
}
```

Entrada/Salida. El paquete java.io

- La entrada y salida de datos se refiere a la transferencia de datos entre un programa y los dispositivos
  - de almacenamiento (ej. disco, pendrive)
  - de comunicación
    - con humanos (ej. teclado, pantalla, impresora)
    - con otros sistemas (ej. tarjeta de red, router).
- La **entrada** 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 el paquete **java.io** vamos a tratar la entrada/salida con ficheros.

El paquete java.io

- Está constituido por una serie de interfaces, clases y excepciones destinadas a definir y controlar:
  - el sistema de ficheros,
  - los distintos tipos de flujos
  - y las serializaciones de objetos.

Ficheros

- La forma de mantener información permanente en computación es utilizar ficheros (archivos).
- Un fichero contiene una cierta información codificada que se almacena en una memoria interna o externa como una secuencia de bits.
- Cada fichero recibe un nombre (posiblemente con una extensión) y se ubica dentro de un directorio (carpeta) que forma parte de una cierta jerarquía (ruta, camino o vía de acceso).
- El nombre y la ruta, o secuencia de directorios, que hay que atravesar para llegar a la ubicación de un fichero identifican a dicho fichero de forma unívoca.

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 `File`

- Representa **caminos abstractos** (independientes del S.O.) dentro de un sistema de ficheros
- Un objeto de esta clase contiene información sobre el nombre y el camino de un **fichero** o de un **directorio**.
- Constructores:

```
File(String dir, String nombre)
File(File dir, String nombre)
File(String camino) // incluido nombre
```

- Los objetos de esta clase se pueden crear para directorios y ficheros que ya existan o que no existan

Lectura de fichero (con **`File`** y **`Scanner`**)

1) Crear un **`File`** sobre un nombre de fichero y crear un **`Scanner`** sobre el **`File`** creado

```
Scanner sc = new Scanner(new File("datos.txt"));
```

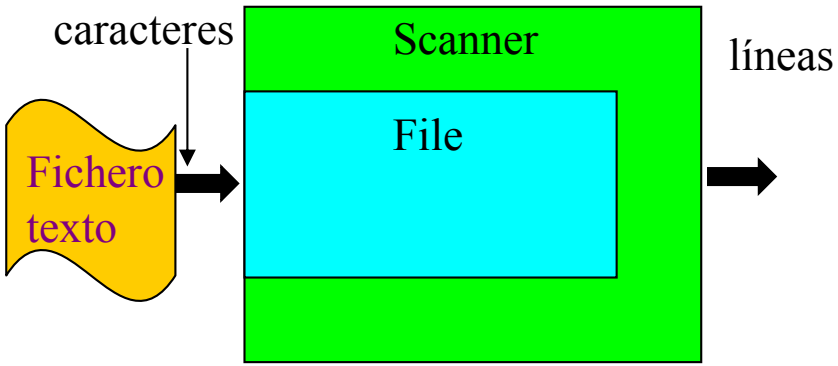
2) Leer líneas con **`hasNextLine()`** y **`nextLine()`** las veces que se necesite

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

3) Cerrar el **`Scanner`**

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

Si se crea en try no hay que cerrarlo



Ejemplo

```
import java.io.*;
public class Ejemplo {
    public static void main(String[] args) throws IOException {
        if (args.length == 0) {
            System.out.println("ERROR: falta el nombre del fichero");
        } else  {
        // leer el fichero de palabras y mostrarlas en pantalla línea a línea
            Scanner sc = new Scanner(new File(args[0]));
            while (sc.hasNextLine()) {
                System.out.println(sc.nextLine());
            }
            br.close();
        }
    }
}
```

Ejemplo

```
import java.io.*;
public class Ejemplo {
    public static void main(String[] args) throws IOException {
        if (args.length == 0) {
            System.out.println("ERROR: falta el nombre del fichero");
        } else  {
        // leer el fichero de palabras y mostrarlas en pantalla línea a línea
            try (Scanner sc = new Scanner(new File(args[0]))) {
                while (sc.hasNextLine()) {
                    System.out.println(sc.nextLine());
                }
            }
        }
    }
}
```

Mejor así (con try).
Cierre automático

Ejemplo

```
import java.io.*;
import java.util.*;
public class Ejemplo {
    public static void main(String[] args) {
        // leer el fichero de palabras y mostrarlas en pantalla línea a línea
        try (Scanner sc = new Scanner(new File(args[0]))) {
            while (sc.hasNextLine()) {
                System.out.println(sc.nextLine());
            }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("ERROR: falta el nombre del fichero");
        } catch (IOException e) {
            System.out.println("ERROR: no se puede leer del fichero");
        }
    }
}
```

Mejor todavía (con try y capturando las excepciones)

La clase `PrintWriter`

- Permite escribir objetos y tipos básicos de Java sobre flujos de salida de caracteres

- 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(...)
```

- Sus métodos no lanzan **`IOException`**

Escritura sobre un fichero de texto

1) Crear un **`PrintWriter`** sobre un nombre de fichero

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

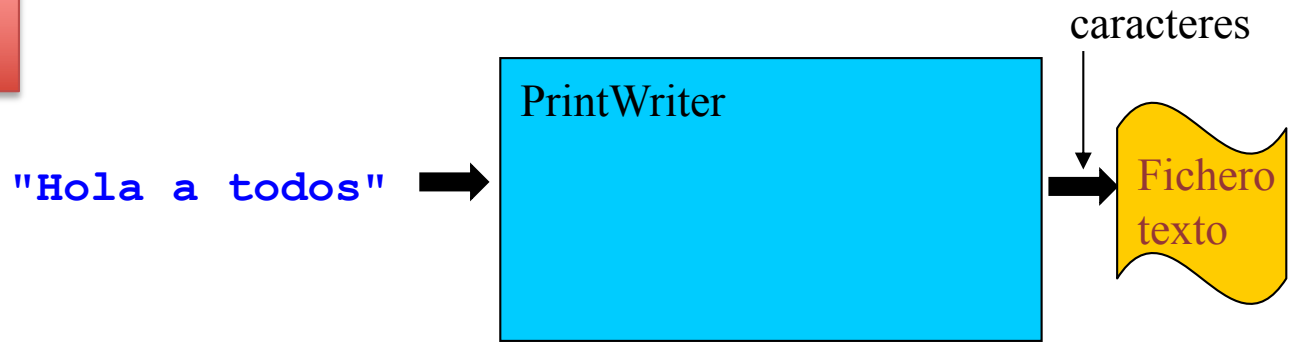
2) Escribir sobre el **`PrintWriter`**

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

3) Cerrar el **`PrintWriter`**

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

Si se crea en try no hay que cerrarlo



Ejemplo

```
import java.io.*;
public class Ejemplo {
    public static void main(String[] args) throws IOException {
        if (args.length == 0) {
            System.out.println("ERROR: falta el nombre del fichero");
        } else  {
            // crear un fichero de palabras
            PrintWriter pw = new PrintWriter(args[0])) ;
            pw.println("amor roma mora ramo");
            pw.println("rima mira");
            pw.println("rail liar");
            pw.close();
            }
        }
    }
}
```

Ejemplo

```
import java.io.*;
public class Ejemplo {
    public static void main(String[] args) throws IOException {
        if (args.length == 0) {
            System.out.println("ERROR: falta el nombre del fichero");
        } else  {
            // crear un fichero de palabras
            try (PrintWriter pw = new PrintWriter(args[0])) {
                pw.println("amor roma mora ramo");
                pw.println("rima mira");
                pw.println("rail liar");
            }
        }
    }
}
```

Mejor así (con try).
Cierre automático

Ejemplo

```
import java.io.*;
public class Ejemplo {
    public static void main(String[] args)  {
        // crear un fichero de palabras
        try (PrintWriter pw = new PrintWriter(args[0])) {
                pw.println("amor roma mora ramo");
                pw.println("rima mira");
                pw.println("rail liar");
        } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("ERROR: falta el nombre del fichero");
        } catch (IOException e) {
                System.out.println("ERROR: no se puede escribir en el fichero");
        }
    }
}
```

Mejor todavía (con try y capturando las excepciones)