Clases Básicas Predefinidas y entrada/salida

Contenido

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

API (Application Programming Interface)

- API es una biblioteca de paquetes que se suministra con la plataforma de desarrollo de Java (J2SDK).
- Estos paquetes contienen interfaces y clases diseñados para facilitar la tarea de programación.
- Los paquetes más básicos son: **`java.lang`** y **`java.util`**.

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`
  - 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:

- **`boolean equals(Object)`**
- **`String toString()`**
- **`int hashCode()`**
- ... consultar la documentación.

El método **equals()**

Antes de Java 16

- 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.

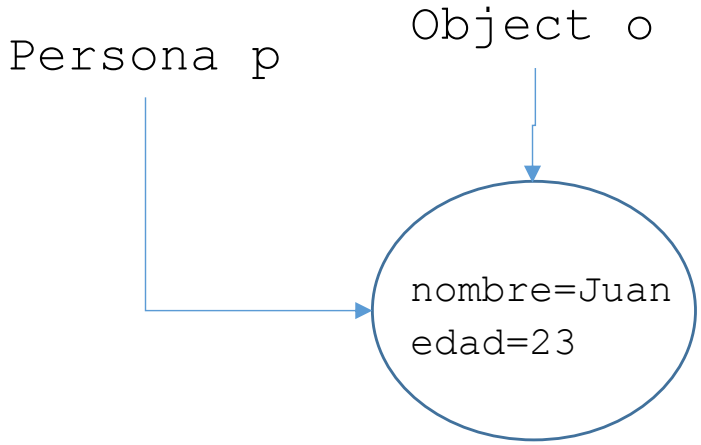
```
class Persona {
	private String nombre;
	private int edad;

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

El método **equals()**

Antes de Java 16

- 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;
        }
        @Override
        public boolean equals(Object o) {
                return (o instanceof Persona)
                        && (((Persona)o).edad == edad)
                        && (((Persona)o).nombre.equals(nombre));
        }
}
```

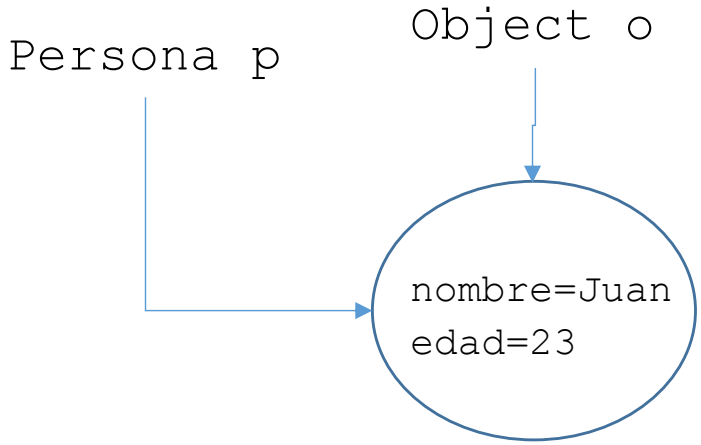


El método `equals()`

Desde Java 16

- 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;
        }
        @Override
        public boolean equals(Object o) {
                return (o instanceof Persona p)
                        && (p.edad == edad)
                        && (p.nombre.equals(nombre));
        }
}
```



p referencia al mismo objeto que o pero visto como Persona (*variable de enlace*)

`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.

```
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 p)
                        && (edad == p.edad)
                        && (p.nombre.equals(nombre));
        }
        @Override public int hashCode() {
                return nombre.hashCode() + Integer.hashCode(edad);
        }
}
```

`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.

```
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 p)
                                && (edad == p.edad)
                                && (p.nombre.equals(nombre));
        }
        @Override public int hashCode() {
                return Objects.hash(nombre, edad);
        }
}
```

Objects está en java.util

`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.

```
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 p)
                                && (edad == p.edad)
                                && (p.nombre.equalsIgnoreCase(nombre));
        }
        @Override public int hashCode() {
                return Objects.hash(nombre.toLowerCase(), edad);
        }
}
```

Objects está en java.util

Clases record (ya vistas)

En Java 16

- Simplifican la creación de clases simples:
  - Son inmutables (sus variables de instancia no se pueden modificar)
  - Constructor con todas sus variables de instancia
  - Métodos de acceso a sus variables de instancia (igual al nombre de la variable)
  - **Nuevo**: equals y hashCode estructural y compatible con todas sus variables de instancia
  - toString que muestra todas sus variables de instancia
  - Se pueden redefinir estos métodos y crear nuevos

nombre y edad son variables de instancia de Persona

```
record Persona(String nombre, int edad){}
record Tuple3<A,B,C>(A _1, B _2, C _3){}

    Persona p1 = new Persona("Juan", 13);
    System.out.println(p1);
    System.out.println(p1.edad());

        Persona[nombre=Juan, edad=13]
        13
```

Clases record (ya vistas)

En Java 21

- Pueden usarse patrones con las clases record en instanceof:

```
record Persona(String nombre, int edad){}
record Tuple2<A,B>(A _1, B _2){}

Persona p = new Persona("Juan", 45);
if (p instanceof Persona(String n, int e)) {
     System.out.println(n);
     System.out.println(e);
}


Tuple2<String, Persona> t = new Tuple2<>("hola",p);
if (t instanceof Tuple2(String s, Persona(String n, Integer e)) {
     System.out.println(s);
     System.out.println(n);
     System.out.println(e);
}
```

Clases record (ya vistas)

En Java 21

- También pueden usarse patrones con las clases record en switch:

```
record Persona(String nombre, int edad){}
record Tuple2<A,B>(A _1, B _2){}


Persona p = new Persona("Juan", 45);
Tuple2<String, Persona> t = new Tuple2<>(”madrid",p);

switch (t) {
    case Tuple2(String s, Persona(String n, Integer e)): {
        System.out.println("Hola " + n);
        System.out.println(s);
        System.out.println(e);
    }
}
```

El método **equals() y record**

En Java 21

- Compara dos objetos de la misma clase.
- Esta predefinido en los record.
  - Con == para tipos básicos y equals para objetos
- Puede redefinirse. Por ejemplo, en Persona queremos que los nombres se comparen independiente de mayúsculas y minúsculas (equalsIgnoreCase)

```
record Persona(String nombre, int edad) {
      @Override
      public boolean equals(Object o) {
            return (o instanceof Persona(String pnombre, int pedad)
                  && (edad == pedad)
                  && (nombre.equalsIgnoreCase(pnombre));
      }
}
```

La clase **`System`**

- Maneja particularidades del sistema.
- Tres variables de clase (static) públicas:
  - **`PrintStream out, err`**
  - **`InputStream in`**
- Métodos de clase (static) públicos:
  - **`void exit(int)`**
  - **`long currentTimeMillis()`**
  - **`long nanoTime()`**
  - **`void gc()`**
  - …
- Consultar documentación para más información.

La clase **Math**

- Incorpora como *métodos de clase* (static), constantes y funciones matemáticas:
  - Constantes
    - **double E, double PI**
  - Métodos de clase:
    - **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 dos clases incluidas en **`java.lang`**:
  - **`String`** - para cadenas constantes
  - **`StringBuilder`** - para cadenas modificables

La clase **`String`**

- 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!";
```

- Las cadenas de los objetos **`String`** no pueden modificarse (crecer, cambiar un carácter, ...).
- Una variable **`String`** puede recibir valores distintos.
- Se pueden crear cadenas de bloques (Java 15)

```
String ss = """
            Esto es una cadena
            de bloques
            permitida""";
```

Métodos de la clase **`String`**

- Métodos de consulta:

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

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

```
substring(int posini, int posfin+1)
substring(int posini)
toUpperCase()
toLowerCase()
static format(String formato,...)
```

- Comparación:

```
compareTo(String str)  // -, 0 ó +
compareToIgnoreCase(String str)  // -, 0 ó +
```

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. Método toString

```
@Override
public String toString() {
    StringBuilder sb = new StringBuilder("[");
    for (int i = 0; i < numLibros; i++) {
        sb.append(libros[i]);
        if (i < numLibros - 1) {
            sb.append(",");
        }
    }
    sb.append("]");
    return sb.toString();
}
```

Expresiones regulares

- Permiten identificar patrones.
- Son muy útiles para expresar formatos.
- La clase `String` tiene dos métodos interesantes:
  - Para verificar que una cadena tiene un formato dado:

```
boolean matches(String exprReg)
```

  - Para extraer datos según unos delimitadores:

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

http://regexr.com https://regex101.com

Expresiones regulares

Cuantificadores

`X?`	X, una vez o ninguna
`X*`	X, cero o mas veces
`X+`	X, una o mas veces

Operadores lógicos

`XY`	X seguido de Y
`X|Y`	O bien X o bien Y

Los paréntesis sirven para agrupar

Caracteres

`[abc]`	a, b o c
`[^abc]`	cualquier caracter excepto a, b o c (negación)
`[a-zA-Z]`	desde a hasta z o desde A hasta Z inclusive (rango)
`[a-d[m-p]]`	desde a hasta d o desde m hasta p: [a-dm-p] (unión)
`[a-z&&[def]]`	d, e o f (intersección)
`[a-z&&[^bc]]`	desde a hasta z excepto b y c: [ad-z] (sustracción)
`[a-z&&[^m-p]]`	desde a hasta z pero no desde m hasta p: [a-lq-z] (sustracción)

Expresiones regulares

Operadores

`X{n}`	X exactamente n veces
`X{n,}`	X al menos n veces
`X{n,m}`	X al menos n veces pero no mas de m veces

MetaCaracteres

`.`	Cualquier caracter
`\d`	Un dígito [0-9]
`\D`	Un no dígito [^0-9]
`\s`	Un espacio [\t\n\x0B\f\r]
`\S`	Un no espacio [^\s]
`\w`	Una letra, _ o dígito [a-zA-Z_0-9]
`\W`	Una no letra, _ o dígito [^\w]

Al incluirlo en una cadena de caracteres, las \ deben duplicarse y los caracteres especiales deben precederse de \\.

Expresiones regulares. Ejemplos

```
[a-d]*                  // aaabadddb, aaaa, d
[ab]{3}(c|d)            // abac, abbd, aaac, bbbd, aaad
[a-z]{1,4}\d            // a1, zz2, dbs4, hhsd9
```

Ejemplos útiles;

DNI: `[0-9]{8}[A-Z&&[^IOU]]`

Fecha simple: `\d{1,2}-\d{1,2}-\d{4}`

Fecha completa:

```
([0-9]|0[1-9]|[1-2][0-9]|3[0-1])(-|/)([0-9]|0[1-9]|1[0-2])(-|/)\d{4}
```

Número decimal:

```
(\+|-)?[0-9]+\.?[0-9]*((e|E)(\+|-)?)?[0-9]*
```

Expresiones regulares. Matches

```
"aaabadddb".matches("[a-d]*");
"abac".matches("[ab]{3}(c|d)");
"hhsd9".matches("[a-z]{1,4}\\d");
```

Ejemplos complejos:

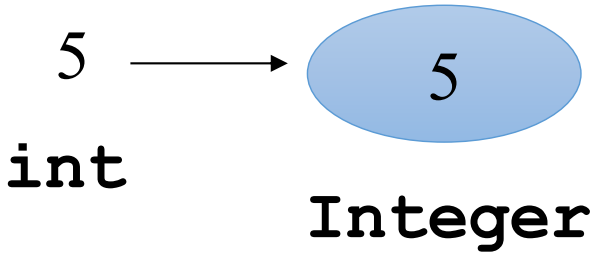
```
"54763381S".matches("[0-9]{8}[A-Z&&[^IOU]]");
"5-10-2014".matches("\\d{1,2}-\\d{1,2}-\\d{4}");
"20/05-2105".matches("([0-9]|0[1-9]|[1-2][0-9]|3[0-
1])(-|/)([0-9]|0[1-9]|1[0-2])(-|/)(\\d{4})");
"-3.45E-2".matches("(\\+|-)?[0-9]+\\.?[0-
9]*((e|E)(\\+|-)?)?[0-9]*");
```

Expresiones regulares. Split

```
String [] items1 = "hola a todos".split("\\s");

items1->{"hola","a","todos"}

String [] items2 =
 "juan garcia;17..,carpintero".split("[;.,]+");

items2->{"juan garcia","17","carpintero"}

String [] items3 ="20/5-2014".split("(-|/)");

items3 -> {"20", "5", "2014"}
```

Las clases envoltorios (*wrappers*)

- Supongamos que tenemos un array de tipo **`Object`**.
- ¿Qué podemos introducir en el array?
  - Sólo objetos. Los tipos básicos no son objetos, por lo que no pueden introducirse en ese array.
  - Para ello se utilizan los envoltorios.
  - En Java 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

- 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");
```

Los envoltorios numéricos

- Métodos para comparar dos números. Devuelve negativo, cero o positivo según el primer argumento sea menor, igual o mayor que el segundo:

```
int compare(XXXX d1, XXXX d2)
```

```
int oi = Integer.compare(34, 45);
int od = Double.compare(34.7, 23.6);
```

- Métodos para encontrar el hashCode de un número:

```
int hashCode(XXXX d1)
```

```
int oi = Double.hashCode(34.23);
```

El envoltorio **Character**

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

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

- Métodos 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 convertir caracteres:

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

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

El paquete **`java.util`**

- Contiene clases de utilidad
  - Las colecciones. (las veremos en el siguiente tema)
  - La clase **`Objects`**.
  - La clase **`StringJoiner`**.
  - La clase **`Optional`.**
  - La clase **`Scanner`**.
  - La clase **`Random`**.
  - Interfaces y excepciones.
  - … consultar la documentación.

La clase **`Objects`**

- Contiene algunos métodos de utilidad. En particular, el método **`hash`**

```
public int hashCode() {
    return nombre.hashCode() + Integer.hashCode(edad);
}
```

- Mejor que sumar dos hashCode es hacer

```
public int hashCode() {
    return Objects.hash(nombre, edad);
}
```

- El método **`hash`** admite un número variable de argumentos.

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();    // "[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;
    return (i < datos.length) ? Optional.of(datos[i]) : Optional.empty();
}
```

```
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 **`mdRecta`**

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

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

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

La clase Scanner

- Permite el análisis sintáctico de cadenas de caracteres.
- Utiliza el espacio en blanco como separador por defecto, y puede utilizar expresiones regulares.
- Puede producir datos primitivos y **`String`** a partir de las cadenas analizadas.
- Se pueden construir objetos Scanner sobre objetos **`String`** y sobre objetos de otras clases de entrada que veremos más adelante. En particular puede usarse con System.in.
- Métodos de instancia:

```
boolean hasNext()
boolean hasNextLine()
boolean hasNextXxxx()
String next()
String nextLine()
Xxxx nextXxxx()
Scanner useDelimiter(String delimitadores)  //expresiónRegular
Scanner useLocale(Locale loc)
void close()
...
```

donde Xxxx es Float, (Integer)Int, Double, Long, Short, Boolean, etc

La clase Scanner con System.in

```
import java.util.*;
public class MainIn {
	public static void main(String[] args) {
		// ------ Uso de la clase Scanner ------------------
		System.out.print("Introduce tu primer apellido y tu edad: ");
		Scanner sc = new Scanner(System.in);
		String apellido = sc.next();
		int edad = sc.nextInt();
		System.out.println("Datos leídos:");
		System.out.println("Apellido: " + apellido +
				"\t" + "Edad: " + edad);
	}
}
```

La clase Scanner con String

- Permite trocear cadenas por delimitadores que pueden ser expresiones regulares. (Igual que split en String)
- El delimitador se indica con useDelimiter(String expR)

`"[,:. ]"`	Exactamente una de entre , : . espacio
`"[,:. ]+"`	Uno o más de entre , : . Espacio

Para que los decimales se interpreten correctamente:

useLocale(Locale.ENGLISH);

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 quiere leer con nextInt() y lo siguiente no es un entero

La clase Scanner sobre un String

```
import java.util.Scanner;

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

Un analizador simple con la clase Scanner

```
import java.util.Scanner;
public class MainAS {
	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);
			   }
			}
		}
	}
}
```

Un analizador simple con la clase Scanner

```
import java.util.Scanner;
public class MainAS2 {
	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("[.]+"); // un punto una o más veces
			while (sc.hasNext()) {
				String datoPersona= sc.next();
				try (Scanner scPersona = new Scanner(datoPersona)) {
					scPersona.useDelimiter("[,:\\-]+");
					// , : o - una o más veces
					String nombre = scPersona.next();
					int edad = scPersona.nextInt();
					Persona persona = new Persona(nombre, edad);
					System.out.println(persona);
				}
			}
		}
	}
}
```

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
double nextGaussian()
…
```

- Consultar la documentación para información adicional.

El paquete **`java.io`** y **`java.nio.file`**

- Estos paquetes proporcionan lo necesario para la gestión de las entradas y salidas de datos de un programa (a través de flujos).

- Están constituidos por una serie de interfaces y clases 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 que forma parte de una cierta jerarquía.
- 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.

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

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

  `Path.of(String fichero)`

```
Path p2 = Path.of("c:/users/juan/datos.txt");
```

- La clase **`Files`** utiliza estos objetos para operar con ficheros o directorios, créalos, borrarlos, saber si existen, obtener información, abrirlos para lectura, etc.

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

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

Lectura de fichero de texto. `Files.readAllLines`

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

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

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

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

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

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

Ejemplo: LeeConFiles

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

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

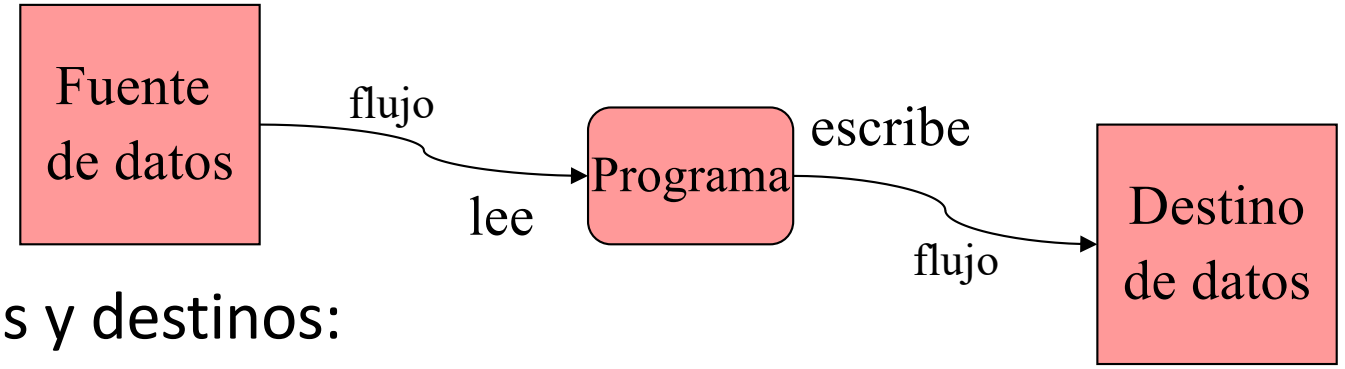
Lee todo el fichero en memoria

Flujos en Java

- En computación, un flujo de datos o **stream** es una secuencia de datos codificados que se utiliza para transmitir o recibir información.
- Java utiliza la noción de flujo como **objeto** que media entre una fuente o un destino y el programa.
- Java distingue entre:
  - **flujos de entrada y flujos de salida** (s/ sentido de circulación)
  - **flujos de texto y flujos binarios** (s/ codif. 16 u 8 bits)
  - **flujos primarios (iniciales) y flujos secundarios (envoltorios)**

Entrada/Salida basada en flujos

- Esquema de funcionamiento:

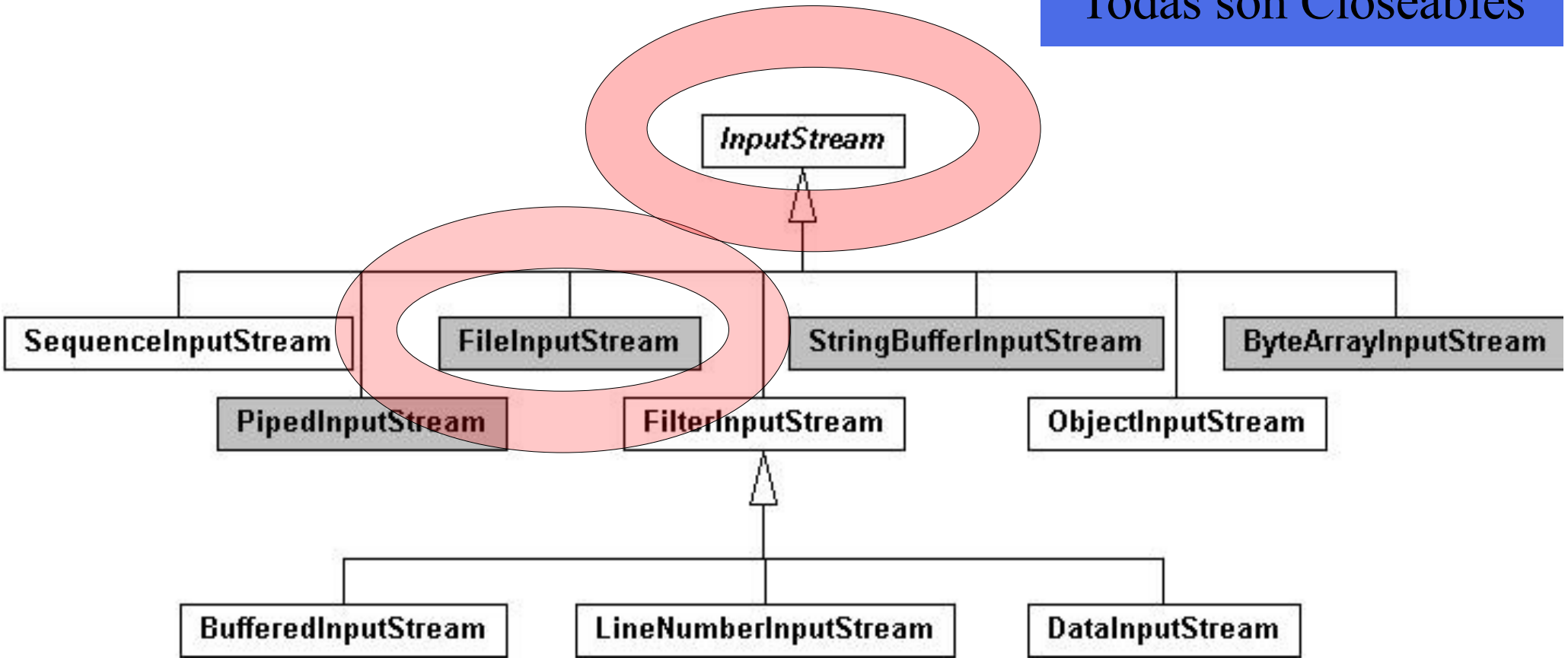


- Fuentes y destinos:
  - array de bytes,
  - fichero,
  - pipe,
  - conexión de red,
  - consola …

(Los flujos siempre conectan con un dispositivo físico)

La familia `InputStream`

Todas son Closeables

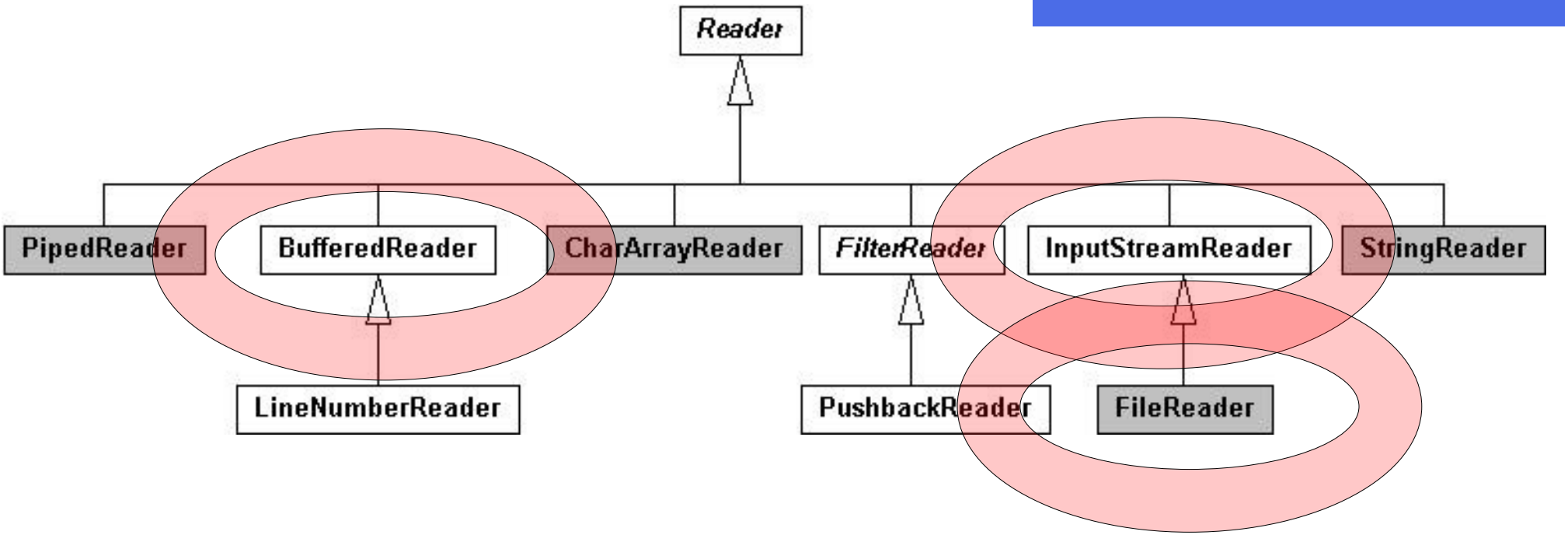


Iniciales

No iniciales

La familia **`Reader`**

Todas son Closeables



Iniciales

No iniciales

BufferedReader

- Proporcionan eficiencia a la hora de leer
  - Utilizan un buffer intermedio
- **BufferedReader**
  BufferedReader(Reader ent)
  - Métodos de instancia

```
String readLine()
boolean ready()
boolean markSupported()
void mark(int readAheadLimit)
void reset()
long skip(long n)
void close()
```

La clase **Files** proporciona métodos para la creación de un **BufferedReader**

```
BufferedReader Files.newBufferedReader(Path)
BufferedReader Files.newBufferedReader(Path, Charset)
```

Ejemplo: LeeBufferedFile

```
import java.io.*;
import java.nio.file.*;

public class LeeBufferedFile {
    public static void main(String[] args) throws IOException {
        String fichero = "personas.txt";
        try (BufferedReader br =
                Files.newBufferedReader(Path.of(fichero))) {
            String linea = br.readLine();
            while (linea != null) {
                System.out.println(linea);
                linea = br.readLine();
            }
        }
    }
}
```

BufferedReader es Closeable

En memoria solo se tiene una línea del fichero cada vez

Lectura de un fichero de texto. Opción Scanner

- Se pueden construir objetos Scanner sobre objetos **`String`**, **`Path`** y otros.
- Métodos de instancia:

```
boolean hasNextLine()
String nextLine()
```

Lectura de fichero de texto

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

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

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

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

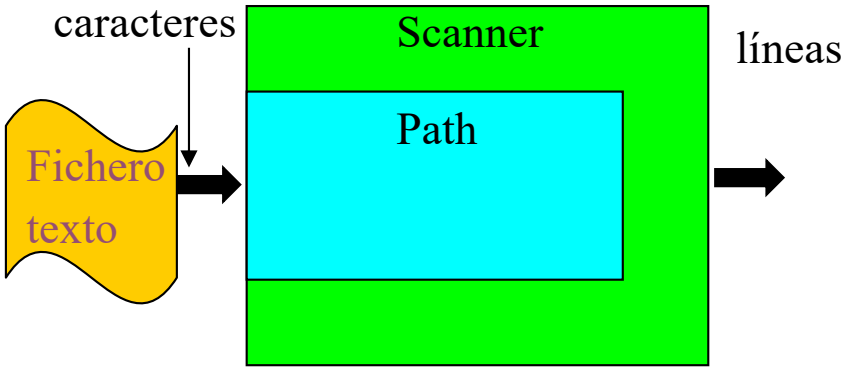
3) Leer líneas con **`hasNextLine()`** y **`nextLine()`** las veces que se necesite

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

4) Cerrar el **`Scanner`**

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

Si se crea con try no hay que cerrarlo



Ejemplo: LeeScanner

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

Scanner es Closeable

En memoria solo se tiene una línea del fichero cada vez

Cuatro opciones para leer un fichero de texto.

1) **`Files.readAllLines`** Se lee todo el fichero en memoria colocando cada línea en una lista de **`String`**.
   1) Crea una **List<String>** con todas las líneas del fichero

2) **`Files.newBufferedReader`** Se leen las líneas del fichero de una en una. Solo hay una línea en memoria en cada ciclo. Es **Closeable**.
   1) Las líneas pueden leerse una a una con el método **readLine()**

3) **Scanner** Se leen las líneas del fichero de una en una. Solo hay una línea en memoria en cada ciclo. Es **Closeable**.
   1) Las líneas pueden leerse una a una con **nextLine()** y **hasNextLine()**

4) **`Files.lines().`** Produce un **`Stream<String>`** sobre las líneas del fichero.

Lectura de fichero de texto

Si hubiera una codificación diferente:

```
Files.readAllLines(Path, CharSet);
Files.newBufferedReader(Path, CharSet);
Files.lines(Path, CharSet);
```

La clase **`StandarCharsets`** tiene las siguientes constantes:

- static final **Charset ISO_8859_1**
  - ISO Latin Alphabet No. 1, also known as ISO-LATIN-1.
- static final **Charset US_ASCII**
  - Seven-bit ASCII, also known as ISO646-US, also known as the Basic Latin block of the Unicode character set.
- static final **Charset UTF_16**
  - Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark.
- static final **Charset UTF_16BE**
  - Sixteen-bit UCS Transformation Format, big-endian byte order.
- static final **Charset UTF_16LE**
  - Sixteen-bit UCS Transformation Format, little-endian byte order.
- static final **Charset UTF_8**
  - Eight-bit UCS Transformation Format.

Por defecto se usa **UTF_8**

```
public class SPersona {

    private List<Persona> personas = new ArrayList<>();

    public void leePersonas(String fichero) throws IOException {
        for (String  linea: Files.readAllLines(Path.of(fichero)))
            personas.add(stringAPersona(linea));
    }

    public Persona stringAPersona(String linea) {
        Persona persona = null;
        try (Scanner scLinea = new Scanner(linea)) {
            scLinea.useDelimiter("[,:\\-]+");
            String nombre = scLinea.next();
            int edad = scLinea.nextInt();
            persona = new Persona(nombre, edad);
        } catch (InputMismatchException e) {
            throw new InputMismatchException("Error de fichero. Número erróneo");
        } catch (NoSuchElementException e) {
            throw new NoSuchElementException("Error de fichero. Faltan datos");
        }
        return persona;
    }

    public List<Persona> getPersonas() {
        return personas;
    }
}
```

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

La clase **`PrintWriter`**

- Permite imprimir representaciones con formato de objetos y tipos básicos de Java sobre flujos de salida de caracteres

```
PrintWriter(String nf)
PrintWriter(Writer sal)
PrintWriter(Writer sal, boolean flush)
PrintWriter(OutputStream sal)
PrintWriter(OutputStream sal, boolean flush)
```

- Métodos de instancia:

Para imprimir todos los tipos básicos y objetos

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

- Sus métodos no lanzan **`IOExeption`**

La clase **`BufferedWriter`**

- Permite imprimir cambiando el juego de caracteres

```
Files.newBufferedWriter(Path)
Files.newBufferedWriter(Path, Charset)
```

- Métodos de instancia:

  Para imprimir todos los tipos básicos y objetos

  `write(String)`

- Lanza **`IOExeption`** si ocurre un error

Escritura de fichero de texto

1) Crear un **`PrintWriter`** directamente sobre un fichero

```
PrintWriter pwF = new PrintWriter("datos.tex");
```

Automáticamente se crea un **`FileOutputStream`** seguido de un **`OutputStreamWriter`** con decodificación por defecto

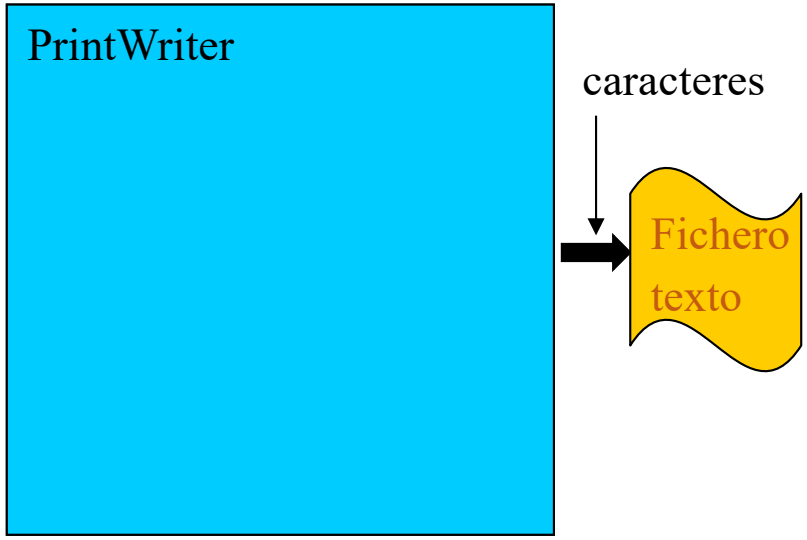
2) Escribir

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

3) Cerrarlo

```
pwF.close();
```

Si se crea con try no hay que cerrarlo



Ejemplo

```
import java.io.*;
public class GuardaPalabras {
    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 el fichero");
        }
    }
}
```

```
import java.nio.Files;
import java.nio.file.Paths;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;
public class EscrituraLectura1 {
     public static void main(String[] args) throws IOException {
          String fichero = "palabras.txt";
          try (PrintWriter pw = new PrintWriter(fichero)) {
               pw.println("amor roma mora ramo");
               pw.println("rima mira");
               pw.println("rail liar");
          }
          // leer el fichero de palabras
          for (String linea :
                         Files.readAllLines(Paths.get(fichero)))

               System.out.println(linea);
     }
}
```

```java
import java.nio.Files;
import java.nio.file.Paths;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Scanner;
public class EscrituraLectura2 {
	public static void main(String[] args) throws IOException {
		String fichero = "palabras.txt";
		// imprimir un fichero de palabras
		try (PrintWriter pw = new PrintWriter(fichero)) {
			pw.println("amor roma mora ramo");
			pw.println("rima mira");
			pw.println("rail liar");
		}
		// leer el fichero de palabras mostrando palabra a palabra
		for (String linea : Files.readAllLines(Paths.get(fichero)))
			try (Scanner scLinea = new Scanner(linea)) {
				while (scLinea.hasNext())
					System.out.println(scLinea.next());
			}
	}
}
```