Clases Básicas Predefinidas

Clases Básicas Predefinidas

- Organización en paquetes
- Clases básicas: java.lang
  - Object, System, Math
- Clases del paquete java.util
  - Cadenas de caracteres, Random
- Clases del paquete java.io
  - File, PrintWriter

2

En este tema se explica la organización en paquetes de Java, y se describen algunas clases características de varios paquetes de Java, como son java.lang, java.útil y java.io. Del primero se verán las clases Object, System, Math y diferentes clases para manejar cadenas de caracteres, como String.

Dentro del paquete java.lang, también se verán una serie de clases denominadas contenedoras, una por cada tipo básico.

Del mismo modo, del paquete java.útil se verán algunas clases como Random o Scanner.

Y, por último, del paquete java.io trataremos clases para manejar la entrada y salida, como File y PrintWriter.

Organización en paquetes (packages)

- 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.

3

Los paquetes permiten agrupar las clases que estén relacionadas de alguna forma.

La librería de clases de Java se organiza en paquetes ya predefinidos.

También se pueden definir paquetes propios.

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.

4

LA definición de un paquete se realiza incluyendo al principio de un fichero la palabra reservada package, seguida del nombre del paquete. Las clases que se definan en ese fichero se considerarán dentro del paquete con esa denominación.

En caso de que no se incluya esta declaración package, las clases que se incluyan en el fichero se considerarán dentro de un paquete anónimo que se genera por defecto.

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

    ```
    sanidad.Medico med;
    ```
  - importarla al comienzo del fichero y usar su nombre simple

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

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

5

Desde fuera de un paquete solo es posible acceder a las clases e interfaces que sean públicas..

Para acceder desde un paquete a una clase o interfaz que pertenece a otro paquete es necesario utilizar el nombre del paquete seguido del nombre de la clase (y separados por un punto). Por ejemplo, para declarar una variable r de la clase Rectangulo del paquete grafico, tendríamos que escribir:

grafico.Rectangulo r;

O bien, hay que importar al comienzo del fichero la clase indicando el paquete:

import grafico.Rectangulo;

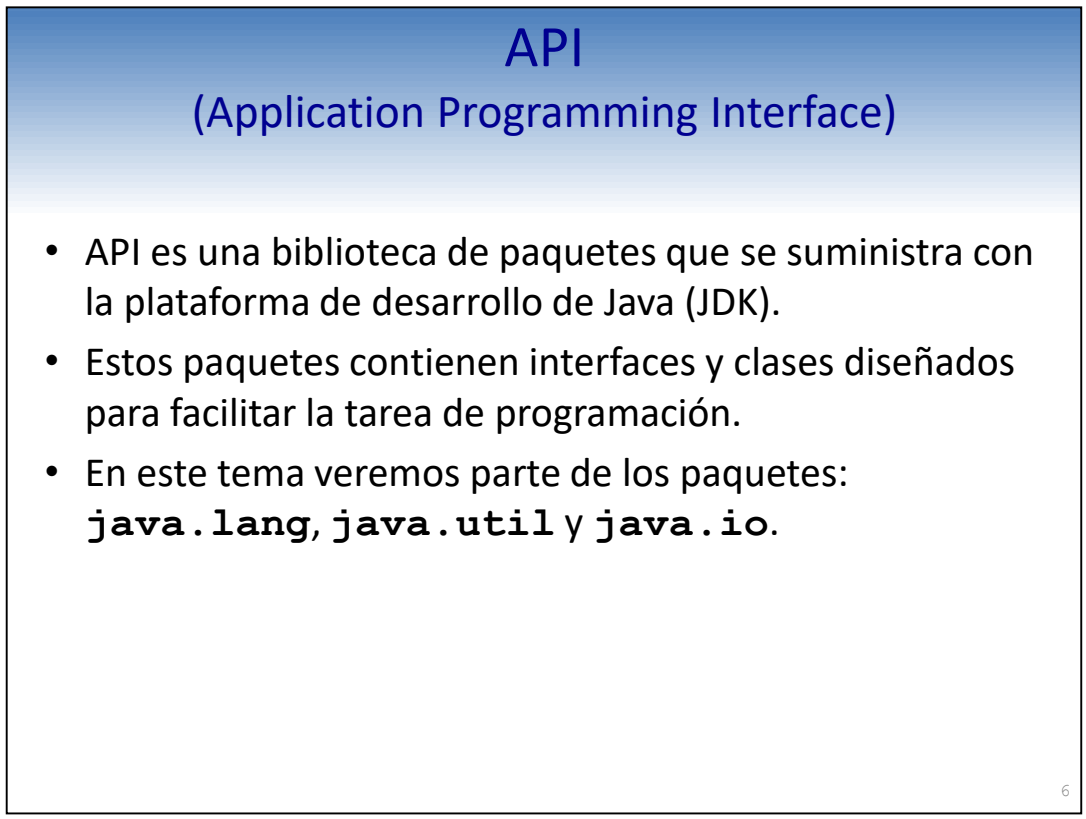
Y después de esa importación, sí se puede mencionar la clase Rectangulo directamente:

Rectangulo r;

Cuando se quieren importar todas las clases que hay un paquete se utiliza el asteristo '*': Por ejemplo,

import grafico.*;

Importa todas las clases del paquete grafico.



La API (Application Programming Interface) es una biblioteca de paquetes que acompaña a cualquier distribución de Java (JDK). Los paquetes de la API incluyen interfaces y clases que aportan funcionalidad variada.

Para acceder a la documentación de la API de Java se puede buscar en línea o utilizar la que proporciona el IDE Eclipse.

Por ejemplo, la de la versión Java 1.8 está accesible en:

https://docs.oracle.com/javase/8/docs/api/

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.

7

El paquete java.lang es el único paquete de la API que no hace falta importar para poder utilizar cualquiera de sus clases.

Algunas clases del paquete son: Object, System, Class, Math, String, etc.

E incluye también algunas interfaces como: Comparable, que veremos más adelante. A modo de adelanto, la interfaz Comparable servirá para caracterizar cuando los objetos de una clase C se pueden comparar (es decir, se puede determinar cuándo una instancia de C es menor que otra). Esto se va a conseguir incluyendo en la interfaz un método

public int compareTo(C objeto)

que devuelve un número positivo cuando el que reciba el mensaje es mayor que el que se pasa como argumento, que devuelve 0 cuando sean iguales, y que devuelva un número negativo cuando el primero sea menor que el segundo. Pero sobre esto volveremos más adelante.

Este paquete también incluye las clases que representa excepciones

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)`**
- **`int hashCode()`**
- **`Object clone()`**
- **`Class getClass()`**
- **`void finalize()`**
- ... consultar la documentación.

8

La clase Object es la clase que está en la raíz de toda la jerarquía de herencia. Es decir, todas las clases heredan de Object (aunque no se indique explícitamente). Es decir, si una clase no hereda (extends) a ninguna otra, es como si heredase de Object.

Esta clase define el comportamiento mínimo común a todos los objetos.

No es un clase abstracta, lo que significa que todos sus métodos tienen alguna implementación que todo objeto heredará si no se redefine.

Algunos de los métodos más característicos de la clase Object son toString, equals, o hashCode.

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 = false;
                if (obj instanceof Persona ) {
                        Persona pers = (Persona) obj;
                        res =  edad == pers.edad &&
                              nombre.equals(pers.nombre);
                return res;
        }
}
```

9

Ya hemos visto el método toString(), que habitualmente redefinimos cuando construimos una clase.

Otro método que es muy relevante es el método equals, que tiene la siguiente signatura:

public boolean equals(Object obj)

La definición que tiene en la clase Object este método coincide con ==. Es decir, si no redefinimos el método equals en nuestras clases, el efecto será idéntico a utilizar ==.

Cuando se quiere definir una relación de igualdad distinta a ==, se redefine el método equals.

Por ejemplo, en el ejemplo que se muestra en la transparencia, se redefine el método equals en la clase Persona, para definir una noción de igualdad entre objetos de tipo Persona, que indica que dos personas son iguales cuando tienen la misma edad y el mismo nombre.

De este modo, si definimos dos objetos Persona de la forma siguiente:

Persona p1 = new Persona(“Pepe”, 20);

Persona p2 = new Persona(“Pepe”, 20);

Si comparamos p1 con p2, el resultado con == y con equals es distintos. Así, p1==p2 devolvería false, mientras que p1.equals(p2) devolvería true.

Obsérvese que la forma de redefinir este método tiene algunas peculiaridades:

- Por un lado, el argumento de equals es de tipo Object, no es de tipo Persona. Esto quiere decir que podemos consultar si una persona es equals a otro objeto que no es Persona (por ejemplo, una jarra). Esto sería perfectamente válido. Por supuesto, lo lógico sería que el resultado fuese false.
- Teniendo en cuenta lo anterior, para poder preguntar por la edad o por el nombre del objeto que se pase como argumento, es necesario asegurarnos que enviamos el mensaje correspondiente a un objeto que sea una persona. De ahí que definamos una variable pers (de tipo Persona) que almacene el objeto que se pasa como argumento (obj) haciendo un casting:

  Persona pers = (Persona) obj;

- Pero esto solo lo podemos hacer cuando sabemos que bj es, efectivamente, un objeto de la clase Persona. Para ello utilizamos la expresión obj instanceof Persona, que devuelve true cuando obj es una instancia de Persona.

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.

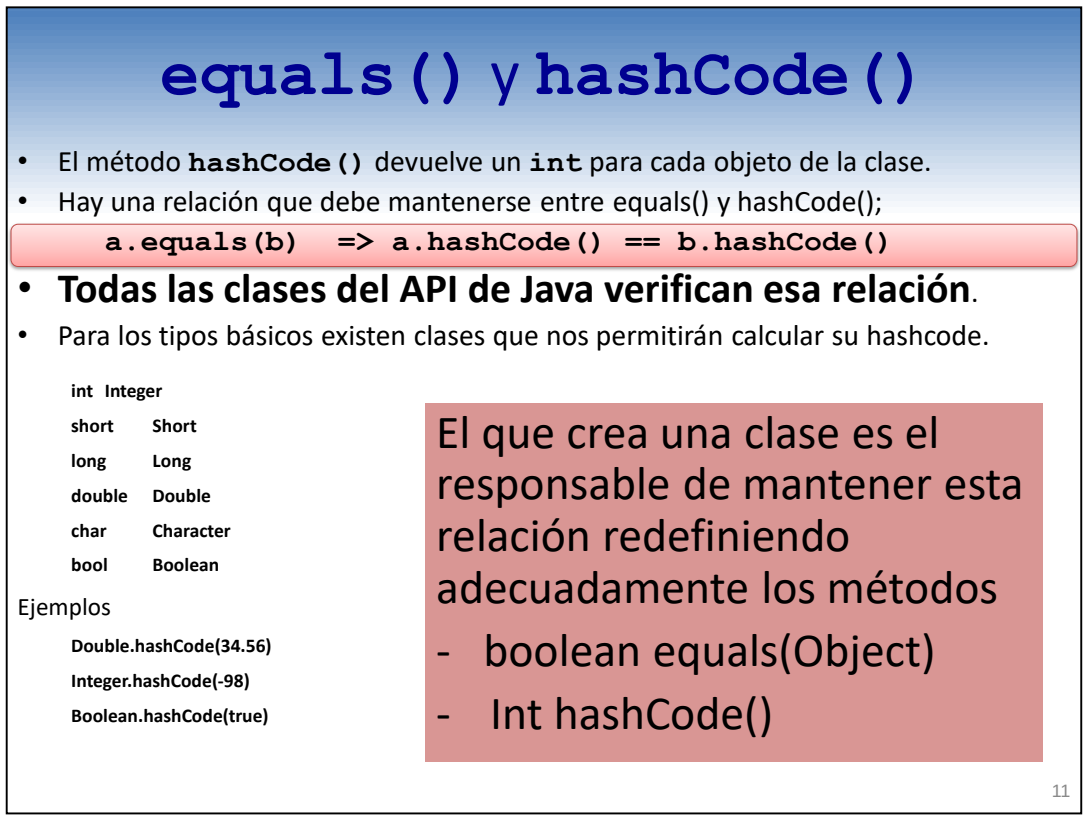
```
class Persona {
        private String nombre;
        private int edad;
        public Persona(String n, int e) {
                nombre = n;
                edad = e;
        }
        public boolean equals(Object obj) {
                boolean esPersona = obj instanceof Persona;
             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

10

Otra forma más compacta y estándar para redefinir el equals es la que se presenta en esta otra transparencia, donde el resultado es equivalente a la versión anterior. En este caso, lo que hacemos es:

1. Definimos una variable booleana esPersona que tomará el valor true si o es un objeto de Persona.
2. A continuación definimos una variable pers de tipo Persona, que será el propio o (pero después de hacer un casting*) o null. Ello dependerá de si el objeto o es Persona.
3. Finalmente, devolveremos el resultado de combinar las condiciones que determinan cuando dos personas son equals. Aquí tenemos que observar que hay un riesgo cuando preguntamos por pers.edad y pers.nombre. Podría ocurrir que pers fuese null (ver el paso 2). Para evitar esta situación, lo que hacemos es preguntar antes (con la conjunción lógica con cortocircuito, &&) por esPersona. Esto significa que si esPersona es false ya no seguimos adelante, y por lo tanto no preguntamos por pers.edad, ni pers.nombre. Solo lo hacemos cuando esPersona es true. Pero en ese caso, no hay problema en preguntar por pers.edad o pers.nombre.

* Obsérvese que el casting que se hace es seguro, porque la asignación pers = (Persona) obj solo se realiza cuando esPersona es true, es decir, cuando o es una instancia de Persona, lo que hace que esta asignación (contraria a la regla del polimorfismo) no dé ningún problema.



Otro método que está totalmente vinculado con equals es hashCode. Es un método que devuelve un entero y no tiene argumentos.

En toda clase en la que se redefina equals debe también redefinirse hashCode, y de forma consistente a la de equals. Esa consistencia se traduce en que si a.equals(b) también debe darse que a.hashCode() sea igual que b.hashCode(). Es decir,

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

La forma de conseguir que esta condición se satisfaga es que los atributos que se utilizan para determinar cuándo dos objetos son equals, se utilicen también para devolver el valor entero de hashCode.

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;
        }
        public boolean equals(Object obj) {
                boolean esPersona = obj instanceof Persona;
                Persona pers = esPersona ? (Persona) obj: null;
                return  esPersona &&
                        edad == pers.edad &&
                        nombre.equals(pers.nombre);
        }
        public int hashCode() {
                return nombre.hashCode() + Integer.hashCode(edad);
        }
}
```

12

Por ejemplo, en el caso que se vio antes de la clase Persona, para garantizar esta consistencia, si dos personas se consideran iguales cuando nombre y edad coinciden, entonces la redefinición de hashCode, se puede definir utilizando un valor entero que dependa de nombre y de edad. La mejor forma de hacerlo es combinar (en este caso mediante una suma) el hashCode() de nombre y de la edad. En el primer caso, como nombre es un String (un objeto), podemos invocar su hashCode() enviándole el correspondiente mensaje: nombre.hashCode(). En el caso de la edad, como es de tipo int (no es un objeto, sino un valor de tipo básico), no se puede invocar directamente enviándole un mensaje. Por eso, lo que se hace es acceder a un método estático hashCode(int) que tiene la clase Integer (más adelante veremos que Integer es una clase envoltorio de int): Integer.hashCode(int).

Así, como vemos en la transparencia, para que p1.equals(p2) sea true, debe ocurrir que p1 y p2 tengan el mismo nombre y la misma edad. En ese caso, nombre.hashCode() e Integer.hashCode(edad) tanto de p1 como de p2, coincidirán. Por eso, una posible redefinición de hashCode() en la clase Persona, es la que se muestra:

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

Porque así se garantiza que p1.hashCode() y p2.hashCode() también coincidan. Es decir, equals y hashCode son consistentes.

La clase `System`

- Maneja particularidades del sistema.
- Tres variables de clase (estáticas) públicas:
  - **`PrintStream out, err`**
  - **`InputStream in`**
- Métodos de clase (estáticos) públicos:
  - **`void exit(int)`**
  - **`long currentTimeMillis()`**
  - **`void gc()`**
  - **`void runFinalization()`**
    provoca la ejecución inmediata de los **`finalize()`** pendientes
  - …
- Consultar documentación para más información.

15

La clase System, también del paquete java.lang, permite controlar diferentes particularidades del sistema.

En particular, incluye tres variables estáticas: una para acceder a la consola de salida, otra a la consola de errores (para poder imprimir mensajes en ellas), y una tercera al dispositivo de entrada (habitualmente el teclado).

Al ser variables estáticas, para referirnos a ellas, utilizaremos:

System.out

System.err

System.in

Esta clase también tiene métodos estáticos públicos que permiten algunas acciones básicas como copiar arrays, obtener la hora actual en milisegundos (desde el 1 de enero de 1970), la invocación del recolector de basura (garbage collector), etc.

Como siempre, para acceder a la documentación de la API de Java se puede buscar en línea. Por ejemplo, la de la versión Java 1.8 está accesible en:

https://docs.oracle.com/javase/8/docs/api/

También se puede acceder a al documentación a través del IDE Eclipse.

La clase **Math**

- Incorpora como *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));`

16

Otra clase muy útil del paquete java.lang ya la hemos utilizado en varias ocasiones. Se trata de la clase Math, donde todos sus métodos y constantes son estáticos y representan operaciones y valores matemáticos.

Es una clase de la que no tiene sentido crear instancias, pues todos sus métodos y constantes se utilizan sobre la propia clase Math.

Por ejemplo, para acceder al valor de PI, escribiríamos Math.PI, o para calcular la raíz cuadrada de un número usaríamos 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 constantes
  - **`StringBuilder`** - para cadenas modificables
  - **`StringBuffer`** - para cadenas modificables (seguras ante tareas)

17

Otra de las estructuras básicas en programación son las cadenas de caracteres. Básicamente, representan una secuencia de caracteres. Para manejar las cadenas de caracteres, Java define varias clases (String, StringBuilder, StringBuffer, entre otras).

Nosotros nos centraremos especialmente en String y StringBuilider. La primera para tratar cadenas de caracteres que no cambian (es decir, no pueden crecer o acortarse) y la segunda clase para manipular objetos que representen cadenas de caracteres que pueden cambiar.

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!";
```

- 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...).

18

La clase String ya la hemos utilizado.

Los objetos de esta clase permiten hacer referencia a cadenas de caracteres que no pueden cambiar.

Por ejemplo, si defino una variable como en la transparencia:

String cadena = "¡Hola!";

No es posible cambiar ningún carácter de la cadena. Es decir, si yo quiero convertir la cadena para que se convierta en "¡Mola!" (por ejemplo), no es posible enviarle a cadena ningún mensaje que cambie el segundo carácter por 'M'. La única forma sería reasignando un nuevo valor a cadena:

cadena = "¡Mola!";

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
```

19

La clase String, entre otros, incluye métodos para:

- Obtener la longitud de una cadena
- Obtener el carácter que ocupa una determinada posición (pero no cambiarlo por otro)
- Obtener la primera o la última posición que ocupa un carácter determinado
- Obtener la primera o la última posición a partir de la cual aparece una subcadena de caracteres

Todos ellos, métodos de consulta, que devuelven algún dato (entero o carácter, en los ejemplos mostrados).

Si se intenta acceder a una posición que no existe en una cadena, por ejemplo:

```
String cad = "¡Hola!";
char car = cad.charAt(6);
```

El sistema lanza una excepción del tipo:

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`

20

Los anteriores no son los únicos métodos, sino que hay una buena cantidad de comportamiento adicional. Aquí se muestran algunos otros que devuelven nuevos objetos String (nuevas cadenas de caracteres), a partir de los string a los que les enviemos esos mensajes.

Por ejemplo, hay varios métodos (substring) que devuelven la subcadena que va de una posición a otra, o la subcadena que empieza en una posición hasta el final. O métodos (replace) que devuelven una cadena nueva en la que se ha reemplazado una subcadena (str1) por otra (str2), etc.

Obsérvese que en todos los casos, lo que se hace es devolver una cadena nueva, no se cambia la cadena que recibe el mensaje. Es decir,

String cadena = “¡Hola!”;

String otraCadena = cadena.replace(“Hol”,”Rem”);

Debe devolver (y asignar a otraCadena) la nueva cadena “¡Rema!”, pero no se modifica el valor de cadena.

Algunos de estos métodos también pueden lanzar la 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

21

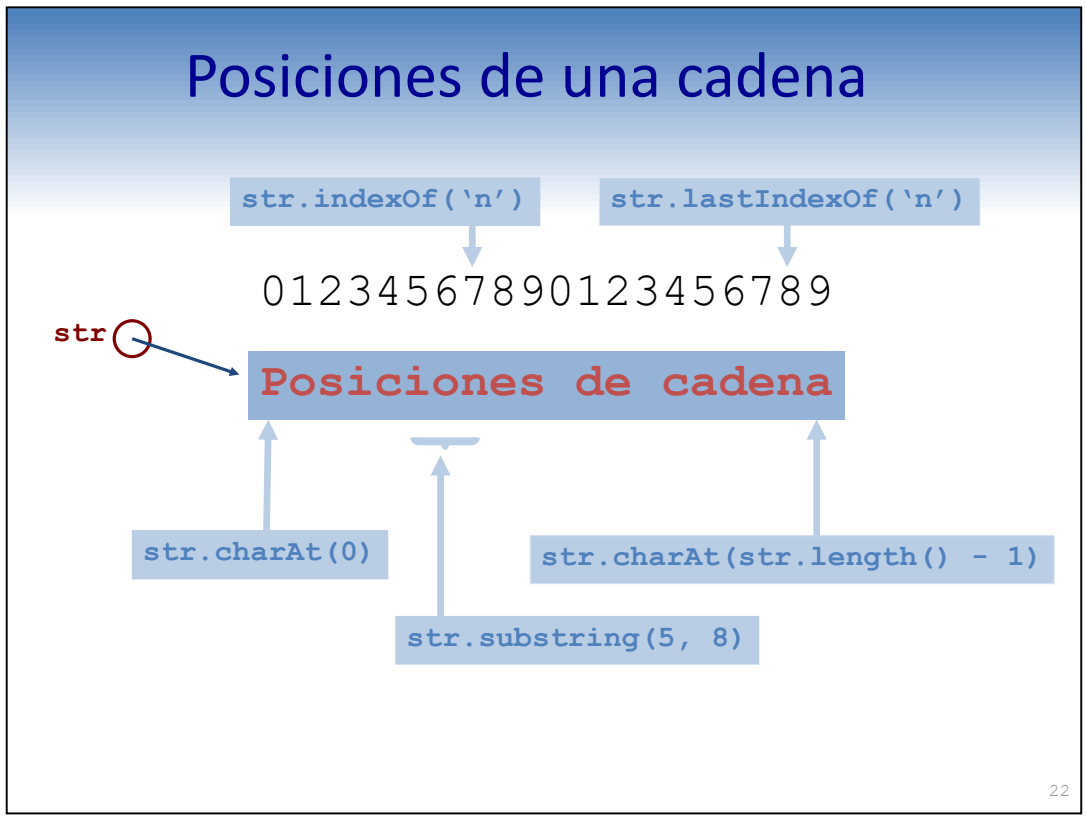
Ya se ha advertido que para comparar cadenas de caracteres es importante utilizar el método equals. Así, si tenemos dos cadenas de caracteres:

```
String cad1 = "Hello!";
String cad2 = cad1.substring(0,4);
```

Si preguntamos por cad2 == “Hell” obtendríamos false, pero si preguntamos por cad2.equals(“Hell”) obtendremos true. El contenido de ambas cadenas cad2 y “Hell” es el mismo, por lo tanto son equals, pero NO es la misma cadena, por lo tanto no son ==.

Otros métodos para comparar cadenas son: equalsIgnoreCase, similar a equals, pero sin tener en cuenta mayúsculas o minúsculas.

El método compareTo determina cuándo una cadena es menor (lexicográficamente) que otra. Devuelve un entero negativo cuando la receptora es anterior lexicográficamente que la que se pasa como argumento, cero si son iguales y un entero positivo si la receptora es posterior (es decir, mayor).



En esta transparencia se ilustra el funcionamiento de varios de los métodos que hemos mencionado sobre un caso práctico.

```
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() {
                …
        }
                …
}
```

23

Esta clase ilustra el uso de otros de los métodos que hemos mencionado.

La clase NombreFichero representa identificadores de fichero, donde distinguimos tres variables de instancia: la cadena con todo el camino para llegar al fichero, el separador que se utiliza para distinguir unas carpetas de otras (puede ser distinto dependiendo del sistema operativo) y el separador utilizado para la extensión del nombre del fichero (también puede variar de un sistema operativo a otro).

Por ejemplo, podríamos crear un objeto de esta clase como:

```
NombreFichero fotoPerfil =
    new NombreFichero("C:/Usuarios/Invitado/Imagenes/miFoto.jpg", '/', '.');
```

A partir del objeto fotoPerfil, podemos obtener la extensión del nombre del fichero, o el nombre del fichero, mediante los métodos que se describen en la transparencia. Así, fotoPerfil.extensión() debería devolver la cadena "jpg" y fotoPerfil.nombre() debería devolver "miFoto".

El método directorio() que debería devolver sobre fotoPerfil la cadena de caracteres "C:/Usuarios/Invitado/Imágenes" se propone como ejercicio.

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 decimales. “%9.2f”
  - b para booleanos “%b”
  - c para caracteres. “%c”
- Se pueden producir las excepciones:
  - MissingformatArgumentException
  - IllegalFormatConversionException
  - UnknowFormatConversionException
  - …

24

El método format se incorpora a la clase String desde la versión 1.5 de Java.

Es un método estático que tiene un número indeterminado de argumentos. El primer argumento es una cadena de caracteres que puede incluir indicadores de la forma % seguido del tipo de datos esperado. El resto de argumentos de format son los datos que ocupan el lugar del indicador para construir la cadena de caracteres que devuelve como resultado el método format.

Los formatos que se aplican con % son:

- s para cualquier objeto (en realidad se aplica toString())
- d para números sin decimales
- f para números decimales
- b para booleanos
- c para caracteres

Así, si cadena = “¡Hola!”, la expresión:

String.format(“La cadena %s mide %d”, cadena, cadena.length());

Devuelve la cadena de caracteres siguiente:

“La cadena ¡Hola! Mide 6”

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
```

25

La combinación de format y print se puede conseguir con el método printf de las clases PrintStream y PrintWriter. En particular, sobre el objeto System.out es posible utilizar el método printf, que es equivalente a utilizar format y luego println, del modo siguiente.

String s = String.format(...);

System.out.println(s);

Es equivalente a:

System.out.printf(…);

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"}
```

26

Otro método muy útil de la clase String es el método split:

String[] split(String expresión)

donde el argumento es una expresión que define una serie de delimitadores, de forma que el resultado es un array de String, donde cada elemento del array es una parte (token) en los que se puede dividir el String que reciba el mensaje, atendiendo a los delimitadores.

Por ejemplo, consideramos como delimitadores la expresion "[,.¨:]", estamos considerando que cada aparición de un espacio, una copa, un punto, un punto y coma o dos puntos, define un token distinto. Así:

String[] items1 = "hola.a to:dos".Split("[,.;:]");

Devolvería en items1 el array formado por {"hola", "a", "to", "dos"}

Si, lo que utilizamos como expresión para delimitadores es "[,.¨:]+", el símbolo + final indica que una o varias apariciones de alguno de los símbolos (espacio, coma, punto y coma o dos puntos) se considera un separador. Es decir,

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

Devolvería en items2 el array formado por {"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.
- En versiones anteriores del JDK se usaba **StringBuffer** en lugar de **StringBuilder**.
- **StringBuffer** se diferencia de **StringBuilder** en que la primera es segura frente a tareas.

27

La clase StringBuilder es similar a String, pero permite que las cadenas se amplíen, reduzcan y se modifiquen.

Métodos de la clase **StringBuilder**

- Métodos de consulta:

```
length()
capacity()
charAt(int pos)
```

- Métodos para construir objetos **String**:

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

- Métodos para modificar objetos **StringBuilder**:

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

28

Los métodos de StringBuilder son similares a los de la clase String, con métodos para consultar datos y para construir nuevos String, y además incluye métodos para modificar los objetos StringBuilder.

Así, por ejemplo, el método append(String str) permite añadir una cadena de caracteres a otra que sea un StringBuilder.

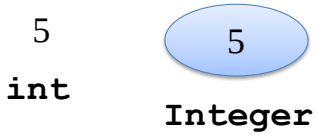
```
public class StringDemo {
  public static void main(String[] args) {
      String cadena = "Aarón es Nombre";
      int long = cadena.length();
      StringBuilder réplica = new StringBuilder(long);
      char c;
      for (int i = 0; i < long; i++) {
        c = cadena.charAt(i);
        if (c ==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

29

En este ejemplo se ilustra el uso de la clase StringBuilder. Obsérvese cómo la variable réplica va construyéndose incrementalmente a partir de un StringBuilder inicialmente vacío (pero con una longitud máxima coincidente con la longitud de la cadena original), añadiéndole los caracteres de la cadena original. Cambiando el carácter ‘A’ por ‘V’ y ‘N’ por ‘H’.

Las clases envoltorio (*wrappers*)

- Supongamos que tenemos un array de tipo **`Object`**: 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.
  - 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

30

Ya hemos explicado que los tipos básicos tienen una naturaleza distinta a las clases, de forma que los valores de tipo básico no son objetos, ni se pueden tratar como tales. Por ejemplo, no es posible enviar un mensaje a un valor de tipo básico. Las clases denominadas envoltorio permiten dar una representación de objeto a los valores de tipo básico. Así, para cada tipo básico, existe una clase envoltorio. Por ejemplo, Integer para int, Double para double, Character para char, Boolean para boolean, etc

De alguna forma un objeto de una clase envoltorio, encapsula un valor de tipo básico, dándole las características de objeto.

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();
```

- 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 crear envoltorios de números a partir de cadenas de caracteres:

Xxxx valueOf(String)

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

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

31

De las clases envoltorio, si nos centramos en las que son envoltorios numéricos (Integer, Double, Float, Long, Short) todas tienen características comunes.

Incluyen constructores a los que se puede pasar el valor de tipo básico correspondiente o una cadena de caracteres, representando el valor. En ambos casos, se crea un objeto de tipo envoltorio. Así,

new Integer(34)

Devuelve un objeto que encapsula al int 34. Del mismo modo,

new Double("33.56");

Encapsula al valor de tipo double 33.56.

Para cada tipo básico, existe un método que permite obtener el valor que encapsula.

Así, podemos enviar a un objeto de tipo envoltorio mensajes como:

intValue()

doubleValue()

…

En todos los casos, se devuelve el valor de tipo básico que encapsulan: int, double, …

Otros métodos que también incluyen los diferentes envoltorios son parseInt, parseDouble, etc. El uso es:

Integer.parseInt(“234”)

Y lo que devuelve es el entero (int) que se corresponda con la cadena “234”, que es en este caso 234.

Otros métodos, también característicos, son los métodos estáticos valueOf(String) que devuelve el tipo envoltorio correspondiente. Por ejemplo,

Integer.valueOF(“234”)

Devuelve el Integer que capsula al entero 234.

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`**

32

Además de los envoltorios numéricos, tenemos el envoltorio Boolean, que encapsula datos booleanos. Igual que los numéricos, consta de un constructor con un argumento que es una cadena de caracteres representando el valor booleano correspondiente:

new Boolean("false");

Tiene también métodos de instancia como

public boolean booleanValue()

O métodos de clase como:

public boolean parseBoolean(String)

public Boolean valueOF(String)

En los casos donde se espera un String, si este no representa un valor lógico, en vez de producirse un error, lo considera como false.

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 convertir caracteres:

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

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

33

Otro envoltorio es la clase Character, que envuelve a datos de tipo char. Además del constructor habitual Character(char), hay métodos similares a los que hemos visto en otros envoltorios.

Por ejemplo

public char charValue()

Que devuelve el valor char asociado a un tipo evnoltorio.

Además, contamos con métodos estáticos que permiten comprobar si el objeto es:

un dígito (isDigit)

una letra (isLetter)

es minúscula o mayúscuala (isLowerCase, isUpperCase)

También hay métodos de clase para convertir caracteres:

toLowerCase (char)

toUpperCase (char)

Ejemplos de envoltorios

```
int a = Integer.parseInt("34");

Double d = new Double("-45.8989");     // envoltura explícita
double dd = d.doubleValue();

double ddd = Double.parseDouble("32.56");

Long l = Long.valueOf("27.98");        // envoltura implícita

double dddd = dd + d;                  // desenvoltura implícita

Boolean b = (new Boolean("mal")).booleanValue();
```

Se produce la excepción **`NumberFormatException`** si algo va mal en alguna conversión numérica

34

En la transparencia se muestran algunos ejemplos de envoltorio.

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];
```

35

Para facilitar la manipulación de objetos de tipo envoltorio y los correspondientes valores de tipo básico, Java tiene un mecanismo que se denomina boxing y unboxing (envolver y desenvolver). De este modo, cuando se espera un objeto de tipo envoltorio se puede utilizar un valor del tipo básico correspondiente, y al revés, cuando se espera un valor de tipo básico, se puede convertir a un objeto de tipo envoltorio.

Así, por ejemplo, si tenemos un array lista donde el tipo de sus elementos es Double (envoltorio), entonces, aunque lista[i] espere almacenar un Double, podemos asignarle un double (con d minúscula). En la transparencia lo hacemos con 45.5. Es decir, donde se espera un Double, podemos asignar un double. Y esto es igual con todos los tipos envoltorio.

La situación inversa también se da, si sumamos 5.2 y lista[j], aunque el segundo sea un Double (y no un double), se interpreta como el valor de tipo básico, y la suma se realiza de forma correcta.

En el primer caso se ha producido un boxing sobre 45.5, mientras que en el segundo caso se ha producido un unboxing sobre 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 **`Scanner`**.
  - Contiene también interfaces y excepciones
  - … consultar la documentación.

37

Otro paquete que ofrece clases de interés y útiles para la mayoría de las aplicaciones, es el paquete java.útil.

Algunas de las clases que contiene son Random, StringTokenizer y Scanner.

De nuevo, para obtener más información sobre las clases y las interfaces que incluye, es conveniente consultar la documentación de Java.

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.

38

La clase Random permite crear objetos que generan series de números pseudoaleatorios. Existen varios métodos para obtener el siguiente número de la serie, en distintos formatos. Así, podemos pedirle a un objeto de la clase Random, el siguiente float, o el siguiente double, o el siguiente entero (entre 0 y un número dado). Los métodos que permiten acceder a esos números aleatorios son:

float nextFloat()

double nextDouble()

int nextInt(int n)

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).

41

En el paquete java.útil, otra clase de gran interés es la clase Scanner.

Hemos visto hasta ahora cómo realizar la salida sobre la consola (System.out). Básicamente, la forma de emitir información hacia la consola se realiza a través de una variable de clase System.out de tipo PrintStream, y a la que enviamos mensajes del tipo print, y println.

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.

42

Del mismo modo, existe otra variable de clase en System, denominada System.in, que representa el flujo de entrada estándar (texto desde el teclado).

La forma de utilizar System.in no es tan simple como la que se tiene para System.out.

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);
```

43

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`
  - ...

44

Una foma simple para acceder a los datos que puedan llegar desde fuentes de datos diversas (también System.in, pero no solamente) es la clase Scanner.

Esta clase dispone de métodos que permiten leer datos de tipos diferentes.

Un objeto de la clase Scanner se conecta a una fuente de datos y permite leerlos como si se tratase de una secuencia, donde cada dato se distingue del siguiente dependiendo de una serie de "separadores". Por defecto, estos separadores son los espacios, los tabuladores y la línea nueva. Algunos de los métodos que permiten leer datos de tipo diferente son:

- next(), lee y devuelve el siguiente String
- nextLine(), lee y devuelve la siguiente línea completa 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();
            ...
      }
}
```

45

El ejemplo que se muestra en la transparencia crea un objeto (teclado) de la clase Scanner, que se conecta al System.in:

Scanner teclado = new Scanner(System.in);

Sobre ese objeto, se lee primero un String (teclado.next()) y luego un entero (teclado.nextInt()). Cada lectura se hace después de sacar por pantalla (System.out) sendos mensajes, pidiendo introducir el nombre y la edad.

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

46

Cuando se leen datos de un objeto Scanner puede ocurrir que no haya más objetos que leer o que el tipo que se espera no se corresponda con el que llega. En el primer caso, se producirá una excepción del tipo NoSuchElementException y en el segundo caso una InputMismatchException. Esta última se produce, por ejemplo, cuando usamos nextInt() y lo que llega 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`
  - ...

47

Para prevenir situaciones anómalas la clase Scanner proporciona otros métodos que determinan si el objeto Scanner tiene datos de un tipo determinado. Por ejemplo, hastNextDouble() devuelve true si el siguiente dato es un double, o 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?

48

El ejemplo que aquí se presenta considera el mismo objeto

Scanner teclado = new Scanner(System.in);

En el ejemplo, después de sacar por consola el mensaje "Introduzca su edad", se utiliza un bucle (while) que lee el siguiente String de teclado (teclado.next()) y luego pide introducir la edad (mensaje por consola). La condición del bucle while es

! Teclado.hasNextInt()

Esto hace que se siga leyendo mientras no llegue un entero. Una vez que llega un entero por teclado, se lee con nextInt() y se almacena en la variable edad.

El ejemplo ilustra la forma de evitar que se produzca una excepción cuando el usuario no introduce un entero (como se espera), y se sigue intentando la lectura hasta que el usuario introduce un entero.

El ejemplo, no previene la situación en la que se introduzca una edad 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

49

Al inicio de la explicación de la clase Scanner, indicamos que los datos que se iban leyendo con los distintos tipos de mensajes next…, se separaban con delimitadores que, por defecto, eran los espacios, el tabulador y la nueva línea. Estos delimitadores se pueden modificar mediante el método:

useDelimiter(String delmitadores)

Los delimitadores vienen dados por lo que se denominan expresiones regulares.

Por ejemplo, si utilizamos como delimitadores la expresión "[,:.]" estamos considerando cada aparición de coma, dos puntos, punto y espacio, como separadores. Si, por el contrario, utilizamos "[,:.]+" lo que estamos considerando que una o varias apariciones de los símbolos indicados son un separador.

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();
    }
}
```

50

Los objetos Scanner son objetos que necesitan ser cerrados (mediante el mensaje close()) cuando ya no vayan a utilizarse más. Esto es porque la clase Scanner implementa la interfaz Closeable.

En el ejemplo de la transparencia se ilustra cómo después de utilizar el objeto teclado, éste se cierra con close().

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.

53

Aunque hemos ilustrado la funcionalidad de la clase Scanner con la lectura desde teclado, en realidad, se utiliza en muchos otros contextos. De hecho, se puede construir un objeto Scanner que utilice como fuente de datos un String, o como veremos más adelante, ficheros de texto.

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
```

54

En este ejemplo se ve cómo se crea un objeto sc (de tipo Scanner) que tiene como fuente de datos el primer argumento (args[0]) cuando se invoca el Main.

Se consideran como delimitadores la aparición de (uno o varios) símbolos (espacio, punto, coma, punto y coma, guión).

Así, si la ejecución de este programa Main se hace del modo siguiente:

```
java Main "hola  a ; todos. como-estas"
```

Tendríamos: args[0] tomaría el valor "hola a ; todos. como-estas", crearíamos un Scanner (sc) que lo utilizaría como fuente de datos, y dentro del bucle while, leeríamos sucesivos tokens (con sc.next()), para luego imprimirlos. Atendiendo a los delimitadores, la salida sería:

```
hola

a
```

```
todos

como

estas
```

Obsérvese que no ha sido necesario cerrar (con close()) el Scanner sc. Esto es debido a que la creación del Scanner se ha hecho dentro del argumento de try. En esos casos, el cierre del scanner se hace de forma automática.

De hecho, en este ejemplo vemos que try se utiliza solo para evitar el cierre del scanner, puesto que no se hace ningún catch.

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);
				}
			}
		}
	}
}
```

55

Este es otro ejemplo de uso de Scanner sobre un String. En este caso, el String es datos, y contiene información sobre personas, incluyendo nombre y edad.

La forma de distinguir una persona de otra es mediante el uso del punto como delimitador sobre el Scanner sc. Esto permite que leamos y almacenemos en datoPersona la información de cada persona:

```
String datoPersona = sc.next();
```

Una vez leída la información de cada persona (que incluye nombre y edad), la forma de desglosar uno de otra es volviendo a crear un Scanner, scPersona (en esta ocasión utilizando como fuente de datos, el String datoPersona), y definiendo sobre ese Scanner como delimitadores, la coma, los dos puntos y el guión. Si se observa el String datos, son las distintas formas en que se separa el nombre de la edad. Una vez establecido estos delimitadores, no tenemos más que leer mediante next() el nombre y mediante nextInt() la edad. Con estos dos datos, creamos un objeto de la clase Persona:

```
Persona persona = new Persona(nombre,edad);
```

En este ejemplo, lo único que hacemos en imprimir esta información en la consola, pero podríamos hacer cualquier otra cosa, como almacenarla en alguna estructura.

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.

11/04/2020

En el paquete java.io se incluyen clases que permiten gestionar la entrada y salida en Java.

Con entrada y salida nos referimos a la transferencia de datos entre un programa y los distintos dispositivos con los que se puede comunicar. Estos dispositivos pueden ser de almacenamiento (por ejemplo, disco duro, memorias flash), o de comunicación (con personas, a través de teclado, pantalla, impresora, etc.; o con otros sistemas, a través de tarjetas de red, encaminadores, etc.).

La entrada se refiere a los datos que recibe el programa, y la salida a los datos que transmite.

Hemos ya visto la entrada desde teclado y la salida hacia consola.

En el paquete java.io vamos a tratar entrada y salida con ficheros de texto.

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.

Los ficheros o archivos son la entidad básica para mantener información permanente y que se puede almacenar.

Los ficheros contienen información que se codifica (como una secuencia de bits) de alguna forma para almacenarla en memoria interna o externa.

Todo fichero tiene un nombre (que habitualmente posee una extensión que suele determinar su tipo) y se localiza en un directorio (o carpeta) que forma parte de cierta jerarquía de ficheros. Para llegar a un fichero, necesitamos su ruta, camino o vía de acceso, y ello lo identifica de forma unívoca.

Por ejemplo, la ruta para acceder a un fichero puede ser "C:/Usuarios/Administrador/Imágenes/miFoto.jpg".

Los directorios también reciben nombres y se localizan mediante una ruta dentro del sistema de ficheros del correspondiente S.O.

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

Cuando se crea un objeto File no se crea el correspondiente fichero, sólo el acceso a la ruta del fichero o directorio. La idea es que las rutas no dependan del sistema de ficheros.

La clase File tiene varios constructores. El más habitual es el que tiene un argumento de tipo String que determina el camino del fichero, incluyendo el nombre. Pero también hay un constructor con dos argumentos, donde el primero es un String con el directorio que lo contiene, y el segundo el nombre del fichero. Otra posibilidad es utilizar un constructor, también con dos argumentos, donde el primero es un objeto File (indicando un directorio) y el segundo el nombre del fichero dentro de ese directorio.

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"));
```

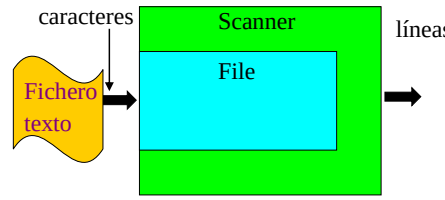
2) Leer líneas con **`hasNextLine()`** y **`nextLine()`** las veces que se necesite

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

3) Cerrar el **`Scanner`**

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

Si se crea en try no hay que cerrarlo



71

Existen muchas formas de realizar lecturas de fichero en Java, pero nos centraremos en una en particular.

Se trata de utilizar un Scanner con un File como fuente de datos. Así,

Scanner sc = new Scanner(new File(“datos.txt”);

Crea un Scanner, sc, que tiene al fichero con nombre “datos.txt” como fuente de datos. El acceso al fichero se consigue con new File(“datos.txt”).

Una vez creado el Scanner, se pueden utilizar sobre él métodos como hasNextLine() y nextLine() para comprobar si hay más líneas en el fichero y poder leerlas, respectivamente.

Para cerrar el Scanner sc se debe utilizar close(). Ahora bien, si el Scanner se creó dentro del argumento de un try, no es necesario cerrarlo con 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
                        Scanner sc = new Scanner(new File(args[0]));
                        while (sc.hasNextLine()) {
                                System.out.println(sc.nextLine());
                        }
                        sc.close();
                }
        }
}
```


En este ejemplo, se supone que pasamos el nombre del fichero como argumento del programa principal. Algo así como:

java Ejemplo “datos.txt”

El programa comprueba si se pasa al menos un argumento. Si no es así (args.length ==0), se imprime un mensaje de error sobre la consola.

En otro caso, se crea el fichero con el nombre que se haya pasado (new File(args[0]) y sobre ese fichero un Scanner, sc.

Después se lee cada línea del fichero, y tan solo se muestra en la consola.

Al final, cerramos el scanner.

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

Si se define la variable Scanner como argumento de try, no es necesario cerrarla

73

Esta otra versión es idéntica a la anterior, con la salvedad que la creación del scanner se hace dentro del argumento del try:

```
try(Scanner sc = new Scanner(new File(args[0])) {
        …
```

La diferencia con la anterior versión es que, en este caso, no es necesario cerrar al final el scanner.

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");
                }
        }
}
```

Capturando excepciones

Y en esta otra versión, además de utilizar el try, usamos las secciones catch para capturar excepciones. La primera excepción se produce cuando al acceder a args[0], ese índice no existe (es decir, no se ha pasado ningún argumento). La segunda se produce cuando hay algún error de entrada/salida. En ambos casos se reacciona sacando un mensaje en consola con indicación de la posible causa del error.

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`**

La clase PrintWriter permite escribir objetos y tipos básicos de Java sobre flujos de salida de caracteres. En particular sobre flujos de salida conectados a un fichero de texto.

La forma de conseguir un objeto PrintWriter conectado a un fichero para guardar datos en él, es mediante el constructor

PrintWriter(String nombreFichero)

Que tiene como argumento una cadena de caracteres que indica el nombre del fichero donde vamos a guardar los datos.

Una vez creado el objeto PrintWriter, si queremos guardar o imprimir datos sobre él, podremos utilizar los métodos ya habituales:

print(….)

println(…)

printf(…)

Estos 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");
```

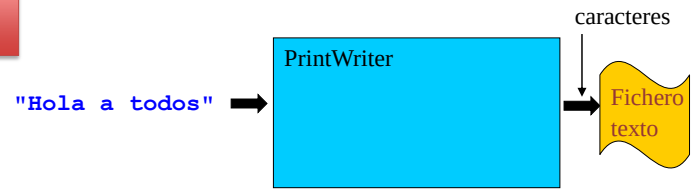
2) Escribir sobre el **`PrintWriter`**

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

3) Cerrar el **`PrintWriter`**

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

Si se crea en try no hay que cerrarlo



77

Por lo tanto, si queremos escribir sobre un fichero de texto, lo que tenemos que hacer es crear un objeto PrintWriter, indicando el nombre del fichero. Por ejemplo,

PrintWriter pw = new PrintWriter(“datos.txt”);

Y para escribir sobre él, podemos utilizar sus métodos print. Por ejemplo,

pw.println(“Esta cadena se guardará en el fichero”);

Por último, es necesario cerrar el PrintWriter, si no se ha creado dentro de un try:

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
                        PrintWriter pw = new PrintWriter(args[0])) ;
                        pw.println("amor roma mora ramo");
                        pw.println("rima mira");
                        pw.println("rail liar");
                        pw.close();
                        }
                }
        }
}
```

78

En este ejemplo, de forma similar al que se usó para ilustrar la lectura desde ficheros, se muestra un ejemplo en el que se escriben varias líneas sobre un fichero.

Igual que en el otro ejemplo, primero se comprueba si se pasa un argumento (con el nombre del fichero). Si no es así, se envía un mensaje de error.

A continuación, se crea el PrintWriter con el fichero como destino, y se imprimen sobre él (se guardan) datos organizados en tres líneas.

Finalmente, se cierra el objeto PrintWriter.

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");
                        }
                }
        }
}
```

Si se define la variable PrintWriter como argumento de try, no es necesario cerrarla

79

En esta segunda versión se hace exactamente lo mismo que antes, pero el PrintWriter se incluye como argumento de try, y por lo tanto no es necesario cerrarlo.

Obsérvese, que el try no tiene ninguna cláusula catch, y solo se ha utilizado para evitar cerrar el PrintWriter.

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");
                }
        }
}
```

Capturando excepciones

Esta tercera versión captura las excepciones IndexOutOfBoundsException e IOException.