Referencia JAVA

Víctor Chico Rodríguez

May 31, 2020

Contents

1 Introducción

Java es un lenguage de programación orientado a objetos, fue desarrollado en 1995 por Sun MicroSystems y está basado en C++, es un lenguaje que se ejecuta en una máquina virtual que interpreta (la JVM, siglas de *Java Virtual Machine*) las instrucciones compiladas a bytecode (el lenguaje de la máquina virtual). Java fue adquirido por Oracle en el año 2010.

1.1 Estructura de un programa

A lo largo de este texto veremos varios ejemplos de código fuente Java así como su salida por pantalla, a continuación un ejemplo de código Java:

```
//Este fichero pertecene al paquete (carpeta) curso.java.manual (curso/java/manual)
package curso.java.manual;

//Esto es una importación de la clase Scanner del paquete java.io
import java.io.Scanner;

//Esto es una clase pública que se llama HolaMundo
public class HolaMundo {

    /*Este es el método main (principal),
es un método especial que servirá como punto de entrada a la aplicación.
Este método es:
- público (puede ser accedido desde cualquier clase)
- estático (puede ser accedido sin necesidad de crear un objeto de la clase)
- no devuelve nada (tipo void)
- recibe como argumentos un array (matriz) de objetos de tipo String (cadena de texto)
*/
    public static void main (String [] args) {
//Este es un objeto de la clase Scanner que se llama scan
//Este objeto se inicializa con la palabra reservada new y
//recibe como argumento System.in (Entrada del sistema)
Scanner scan = new Scanner(System.in);

/*
  Esto es una llamada a un método, concretamente al método print(String)
  del atributo out (que es un objeto de la clase PrintStream)
  de la clase System, este método imprime en la pantalla
```

```
  (consola de texto) el texto que se le pase como parámetro y continúa
  en la misma línea.
  Como parámetro se le pasa la cadena de texto (String) "¿Cómo te llamas?"
  Los valores de tipo String van siempre entre comillas dobles "
 */
System.out.print("¿Cómo te llamas? ");

//Este es un objeto de la clase String (cadena de texto) que se llama nombre.
//Este objeto se inicializa automáticamente con el valor que devuleve
//el método nextString() (Método sin argumentos) del objeto scan.
String nombre = scan.nextString();

/*
  Esto es otra llamada a un método, en este caso al println del atributo
  out de la clase System, nótese la diferencia con la llamada anterior
  (print -- println), ese ln añadido lo que hace es saltar de línea una
  vez haya impreso lo que le pasemos como parámetro.

  En este caso, como parámetro se le pasa una cadena de texto (igual que antes)
  con el valor "Hola, " a lo que le concatenamos (sumamos) el valor de
  la variable nombre
 */
System.out.println("Hola, "+nombre);
    }
}
```

Figure 1: Hola Mundo

Y a continuación un ejemplo de como sería su salida por pantalla si le decimos que nos llamamos Víctor:

```
¿Cómo te llamas? Víctor
Hola, Víctor
```

2 Tipos de datos

Los tipos de datos primitivos en java son los siguientes:

tipo	descripción	clase asociada
byte	número entero de 8 bits (-128 a 127)	Byte
short	número entero de 16 bits (-32768 a 32767)	Short
int	número entero de 32 bits (-2^{32} a 2^{32})	Integer
long	número entero de 64 bits (-2^{64} a 2^{64})	Long
float	número decimal de 32 bits	Float
double	número decimal de 64 bits	Double
boolean	valor booleano o lógico true/false (verdadero o falso)	Boolean
char	caracter de texto (único)	Character

Los tipos de datos normalmente se usan en su forma primitiva (columna tipo) y se pueden asignar directamente, pero a veces es util usar métodos de su clase asociada.

3 Operadores

3.1 Asignación

El operador = se usa para asignar valores a variables:

```
int a = 0;
```

3.2 Aritméticos

En java se pueden realizar multitud de operaciones matemáticas con la misma precedencia que en la vida real, si se necesita modificar se pueden utilizar paréntesis, los operadores aritméticos son los siguientes:

Operador	Descripción
+	Operador de suma
-	Operador de resta
*	Operador de multiplicación
/	Operador de división
%	Operador de resto de la división

El siguiente código es una pequeña demostración de los operadores mencionados:

```
public class Aritmeticos {
```

```
    public static void main (String[] args) {

// Variable de tipo int que tendrá como valor el resultado de 1 + 2
int resultado = 1 + 2;
// El valor de resultado es 3
System.out.println("1 + 2 = " + resultado);
int resultado_original = resultado;

// Los operadores se pueden usar entre variables (numéricas) y números
// en este caso se resta 1 al valor de resultado primero y se asigna a
// la variable resultado después
resultado = resultado - 1;
// El valor de resultado es 2
System.out.println(resultado_original + " - 1 = " + resultado);
resultado_original = resultado;

// Multiplicamos el resultado por 2 y lo volvemos a asignar a la variable
//resultado
resultado = resultado * 2;
// El valor de resultado es 4
System.out.println(resultado_original + " * 2 = " + resultado);
resultado_original = resultado;

// Dividimos el resultado entre 2 y lo asignamos
resultado = resultado / 2;
// El valor de resultado es 2
System.out.println(resultado_original + " / 2 = " + resultado);
resultado_original = resultado;

resultado = resultado + 8;
// El valor de resultado es 10
System.out.println(resultado_original + " + 8 = " + resultado);
resultado_original = resultado;

// Dividimos el resultado entre 7 y nos quedamos con el resto, luego lo
// asignamos
resultado = resultado % 7;
// El valor de resultado es 3
System.out.println(resultado_original + " % 7 = " + resultado);
    }
```

```
}
```

Figure 2: Aritmeticos

```
1 + 2 = 3
3 - 1 = 2
2 * 2 = 4
4 / 2 = 2
2 + 8 = 10
10 % 7 = 3
```

Como vimos anteriormente, el operador suma + se puede utilizar también para concatenar texto:

```
class Concatenacion {
    public static void main(String[] args){
String firstString = "Esto es";
String secondString = " una cadena de texto concatenada.";
String thirdString = firstString+secondString;
System.out.println(thirdString);
    }
}
```

Figure 3: Concatenación

```
Esto es una cadena de texto concatenada.
```

3.2.1 Casteo de valores

Muchas veces, cuando estamos haciendo una operación aritmética, el valor que necesitamos *es más pequeño* que los posibles valores que nos puede dar como resultado la operación, con esto no queremos decir que se produzca si, por ejemplo, sumamos dos números muy grandes (ya que en ese caso, lo que podría ocurrir sería que el valor máximo del tipo de dato se excediera y *diéramos la vuelta*, es decir, nos fuéramos a los números negativos, pero Java no se quejaría de esto), sino que si, por ejemplo, sumamos dos números de tipo *long* y queremos guardar su resultado en una variable de tipo *int*, aunque el valor de los números a sumar cupiera perfectamente en una variable de tipo *int*, Java no tiene forma de saber esto, y se quejará por ello, veamos un ejemplo:

```
class ValorMuyGrande {
  public static void main(String[] args) {
    long n1 = 2;
    long n2 = 3;
    int suma = n1 + n2;
    System.out.println(suma);
  }
}
```

Figure 4: Valor Muy Grande

El resultado que nos daría el compilador sería el siguiente:

```
ValorMuyGrande.java:5: error: incompatible types: possible lossy conversion from long t
  int suma = n1 + n2;
                 ^
1 error
```

Es decir, Java nos está indicando que queremos meter un valor potencialmente más grande que la variable que lo va a contener, se va a producir una *pérdida* en la conversión de `long` a `int`.

En los casos en los que sabemos que ese valor no excederá nuestra variable, tenemos la opción de **castear** (digamos, prometer algo a Java) que la suma devolverá un determinado tipo, esto se hace poniendo entre paréntesis el tipo de dato que devolverá una determinada función, veamos el ejemplo:

```
class ValorCasteado {
  public static void main(String[] args) {
    long n1 = 2;
    long n2 = 3;
    int suma = (int) (n1 + n2);
    System.out.println(suma);
  }
}
```

Figure 5: Valor Casteado

```
5
```

3.2.2 Operadores unarios

En java hay un tipo de operadores aritméticos que sólo se utilizan en un operando, son los operadores unarios:

Operador	Descripción
+	Indica un valor positivo
-	Indica un valor negativo
++	Incrementa en 1 el valor
–	Decrementa en 1 el valor
!	Invierte el valor de un booleano

```
class Unarios {
    public static void main(String[] args) {
int resultado = +1;
// El resultado es 1
System.out.println(resultado);

resultado--;
// El resultado es 0
System.out.println(resultado);

resultado++;
// El resultado es 1
System.out.println(resultado);

resultado = -resultado;
// El resultado es -1
System.out.println(resultado);

boolean exito = false;
// false
System.out.println(exito);
// true
System.out.println(!exito);
    }
}
```

Figure 6: Unarios

```
1
```

```
0
1
-1
false
true
```

Los operadores de incremento y decremento ($++$ y $--$) actuan de manera diferente dependiendo de si se ponen delante o detrás del valor a modificar, si se usan de manera prefija `++variable` el valor se incrementa primero y la variable se usa después (ya incrementada), si se usa de manera postfija `variable++` se utilizará el valor de la variable sin incrementar y luego se incrementará:

```
class PrePost {
    public static void main(String[] args){
int i = 3;
i++;
// imprime 4
System.out.println(i);
++i;
// imprime 5
System.out.println(i);
// imprime 6
System.out.println(++i);
// imprime 6
System.out.println(i++);
// imprime 7
System.out.println(i);
    }
}
```

Figure 7: Prefijos y Postfijos

```
4
5
6
6
7
```

3.3 Lógicos

Son operadores que devuelven valores lógicos (verdadero o falso)

Operador	Descripción
==	igual que
!=	distinto que
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que
&&	Y lógico
||	Ó lógico
instanceof	Objeto pertenece a clase
.equals(objeto)	Método que sirve para comparar objetos

Normalmente estos operadores se utilizarán en sentencias que requieran un valor lógico, como los condicionales o los bucles, de los que hablaremos más adelante, en este ejemplo vemos como, en base a los valores 1 y 2, que operaciones se ejecutan y cuales no:

```
class Comparacion {

    public static void main(String[] args){
int valor1 = 1;
int valor2 = 2;
System.out.println ("valor1="+valor1+", valor2="+valor2);
if(valor1 == valor2) {
    System.out.println("valor1 == valor2 --> " + (valor1 == valor2));
}
if (valor1 != valor2) {
    System.out.println("valor1 != valor2 --> " + (valor1 != valor2));
}
if (valor1 > valor2) {
    System.out.println("valor1 > valor2 --> " + (valor1 > valor2));
}
if (valor1 < valor2) {
    System.out.println("valor1 < valor2 --> " + (valor1 < valor2));
}
if (valor1 <= valor2) {
    System.out.println("valor1 <= valor2 --> " + (valor1 <= valor2));
```

```
}
    }
}
```

Figure 8: Comparación

```
valor1=1, valor2=2
valor1 != valor2 --> true
valor1 < valor2 --> true
valor1 <= valor2 --> true
```

A veces es interesante comprobar si una comprobación cumple mas de una condición o si una sentencia se ejecutará si se cumple alguna de las condiciones posibles, es en este caso que utilizaremos los operadores lógicos `&&` y `||`.

```
class Condicionales {

    public static void main(String[] args){
int valor1 = 1;
int valor2 = 2;
if((valor1 == 1) && (valor2 == 2))
    System.out.println("valor1 es 1 AND (Y) valor2 es 2");
if((valor1 == 1) || (valor2 == 1))
    System.out.println("valor1 es 1 OR (O) valor2 es 1");
    }
}
```

Figure 9: Operadores Condicionales

```
valor1 es 1 AND (Y) valor2 es 2
valor1 es 1 OR (O) valor2 es 1
```

4 Condicionales

En java tenemos principalmente dos estructuras condicionales, la primera es la que se compone con las sentencias `if` y `else`, y la segunda es la sentencia `switch`.

4.1 if y else

La sentencia `if` se escribe de la siguiente manera:

```
if (condicion) {
    proceso;
}
```

Donde `condicion` es un valor booleano (lógico), que puede ser una variable de tipo boolean, un valor `true` o `false` directamente, aunque no tuviera mucho sentido en este caso, o el resultado de una comparación como las que acabamos de ver.

Si la condición se cumple el `proceso` (que puede ser un número indeterminado de sentencias) se ejecuta, si no se cumple, no se ejecuta, decimos que se produce un salto condicional.

Hay veces que queremos que si se cumple una condición se ejecute un determinado código y, si no se cumple, otro, esto lo conseguimos con la sentencia `else` que tiene una forma parecida al `if`, pero en este caso no se especifica condición, sino que la condición es que no se cumpla el `if`.

```
if (condicion) {
    proceso;
} else {
    otroProceso;
}
```

Puede suceder que queramos comprobar una cosa y luego, independientemente otra, en ese caso solo tendríamos que tener un `if` primero y, una vez cerrado, otro con otra condición, en ese caso serían sentencias independientes y no habría ningún problema, pero podemos querer comprobar algo y, si se cumple, otra cosa después, esto lo hacemos *anidando* sentencias `if` o `else`:

```
if (condicion1) {
    proceso1;
    if (condicion2) {
proceso2;
    }
    proceso3;
} else {
    if (condicion3) {
proceso4;
    }
}
```

Si nos fijamos en el `else` (aunque esto puede ocurrir en cualquier otra parte, incluido el bloque del `if`), podemos observar que, en caso de no cumplirse la `condicion1`, podemos tener dentro otra estructura completa de sentencias `if` y cada una puede tener sus respectivos `else` y así indefinidamente, una manera de organizar mejor esté código es utilizando la sentencia compuesta `else if` que nos permite hacer varias comprobaciones sin aumentar el nivel de anidación, por ejemplo:

```
class Elseif {
  public static void main (String [] args) {
    int val = 10;
    if (val == 0) {
      System.out.println("val = 0");
    } else if (val == 1) {
      System.out.println("val = 1");
    } else if (val == 2) {
      System.out.println("val = 2");
    } else if (val == 3) {
      System.out.println("val = 3");
    } else if (val == 4) {
      System.out.println("val = 4");
    } else if (val == 5) {
      System.out.println("val = 5");
    } else {
      System.out.println("val > 5");
    }
  }
}
```

Figure 10: Else-If

```
val > 5
```

En este caso como el valor de la variable `val` es 10, pasaría por cada una de las condicione y, al no cumplirse, entraría por la sentencia `else` si hiciéramos esto anidando sentencias `if` y `else` el código se *iría* muy a la derecha y sería más dificil de leer, pero aún tenemos otra sentencia que nos permite resolver estos problemas de una manera más elegante, la sentencia `switch`.

4.2 switch

El ejemplo anterior, escrito con una sentencia switch sería el siguiente:

```
class Switch {
    public static void main (String [] args) {
      int val = 10;
      switch(val) {
      case 0:
System.out.println("val = 0");
break;
      case 1:
System.out.println("val = 1");
break;
      case 2:
System.out.println("val = 2");
break;
      case 3:
System.out.println("val = 3");
break;
      case 4:
System.out.println("val = 4");
break;
      case 5:
System.out.println("val = 5");
break;
      default:
System.out.println("val > 5");
      }
    }
  }
```

Figure 11: Switch

```
val > 5
```

Como se puede observar, el código es mucho más claro, tenemos una sola sentencia condicional, `switch`, y esta, en base al valor que tenga la variable, entrará por un `case` o por otro y, en caso de que no coincida con ninguno, entrará por el `default`. Si, por ejemplo, cambiásemos el valor de `val` a 3, la salida que nos mostraría el programa sería la siguiente:

```
val = 3
```

Podemos observar también una sentencia que no habíamos visto antes, la sentencia `break`, esta sentencia *rompe* la ejecución del bloque en el que se encuentra, sería como ir a la llave de cierre, normalmente está desaconsejado su uso, pero en la sentencia `switch` es necesaria para cortar la ejecución donde nos interese, ya que, a diferencia de con las estructuras `if-else`, que están englobadas con llaves que nos hacen de corte, los `case` y `default` son etiquetas, y no delimitan código, lo marcan. Veamos que pasa si no ponemos la sentencia `break` en un `switch`.

```
class Switch2 {
    public static void main (String [] args) {
      int val = 2;
      //Inicializamos un contador para saber por cuantos cases pasamos;
      int contador = 0;
      switch(val) {
      case 0:
contador++;
      case 1:
contador++;
      case 2:
contador++;
      case 3:
contador++;
      case 4:
contador++;
      case 5:
contador++;
      case 6:
contador++;
      case 7:
contador++;
      case 8:
contador++;
      case 9:
contador++;
      case 10:
contador++;
System.out.println("He pasado por "+contador+" cases. El número es menor o igual que 1
```

```
      }
    }
  }
```

Figure 12: Switch2

```
He pasado por 9 cases. El número es menor o igual que 10
```

¿Qué ha pasado? El programa ha ejecutado todos los cases uno detrás de otro, ya que ninguno tenía una sentencia `break` para parar la ejecución y ha llegado hasta el último, donde ha imprimido el mensaje. Este ejemplo nos sirve también para ver que la etiqueta `default` no es imprescindible, como en la instrucción `if` no es imprescindible el `else`, simplemente, si no se cumple ninguna de las condiciones contempladas, no se hará nada.

4.3 Condicional ternario

Por último nos queda un último tipo de condicional, llamado ternario o de asignación, esta estructura nos permite asignar un valor a una variable en base al valor de otra y se escribe de la siguiente forma:

```
String miString = (condicion)?"condicion es verdadera":"condicion es falsa";
```

Analizando por partes tenemos, a la izquierda del igual, una declaración de variable de tipo `String` como las que hemos visto hasta ahora, a la derecha tenemos, primero una condición lógica (del mismo tipo que las que se usan en las sentencias `if`, luego un signo de interrogación `?` que es el que nos indica que ese valor lógico no es para asignar a la variable, como hemos visto cuando asignábamos variables de tipo `boolean`, sino que es la condición para asignar la variable, el siguiente valor `"condicion es verdadera"` es el valor que tomará la variable `miString` si `(condicion)` es verdadera. Luego encontramos un signo de dos puntos `:` que separa las condiciones verdadera y falsa y, por último `"condicion el falsa"` que, como se puede intuir, es el valor que tomará `miString` si `(condicion)` es falsa.

Este condicional puede ser escrito con sentencias `if-else` de la siguiente manera (el resultado del código será el mismo):

```
String miString;
if (condicion) {
  miString = "condicion es verdadera";
} else {
```

```
  miString = "condicion es falsa";
}
```

La decisión de usar una u otra dependerá de si se prefiere legibilidad del código (ternaria) o comprensión más visual (if-else).

5 Bucles

La ejecució normal de un programa en java (y en casi cualquier lenguaje de programación) se hace *de arriba a abajo* desde que empieza hasta que termina, los bucles son estructuras de control que permiten que una parte del código se ejecute más de una vez en base a una condición.

5.1 El bucle while

El tipo de bucle más simple que nos encontramos es el bucle `while`, este bucle se va a ejecutar *mientras* (while) la condición se cumpla y, una vez esta deje de cumplirse, seguirá desde el final del mismo.

Es importante que la condición deje de cumplirse en algún momento, y esto es válido para cualquier tipo de bucle, si la condición siempre se cumple decimos que tenemos un bucle infinito, el cual hará que nuestro programa se bloquee.

La estructura de un bucle `while` es la siguiente:

Por ejemplo, si queremos un programa que muestre por pantalla los números del 1 al 10, podemos hacer lo siguiente:

```
class BucleWhile {

    public static void main(String[] args) {
//Ponemos el número con el valor que queremos al principio
int numeroActual=1;

//Bucle while
//Condición: que numeroActual sea menor o igual que 10
while (numeroActual<=10) {

    //Imprimimos por pantalla el número con su valor en este momento
    System.out.println(numeroActual);

    //Aumentamos el valor del número
```

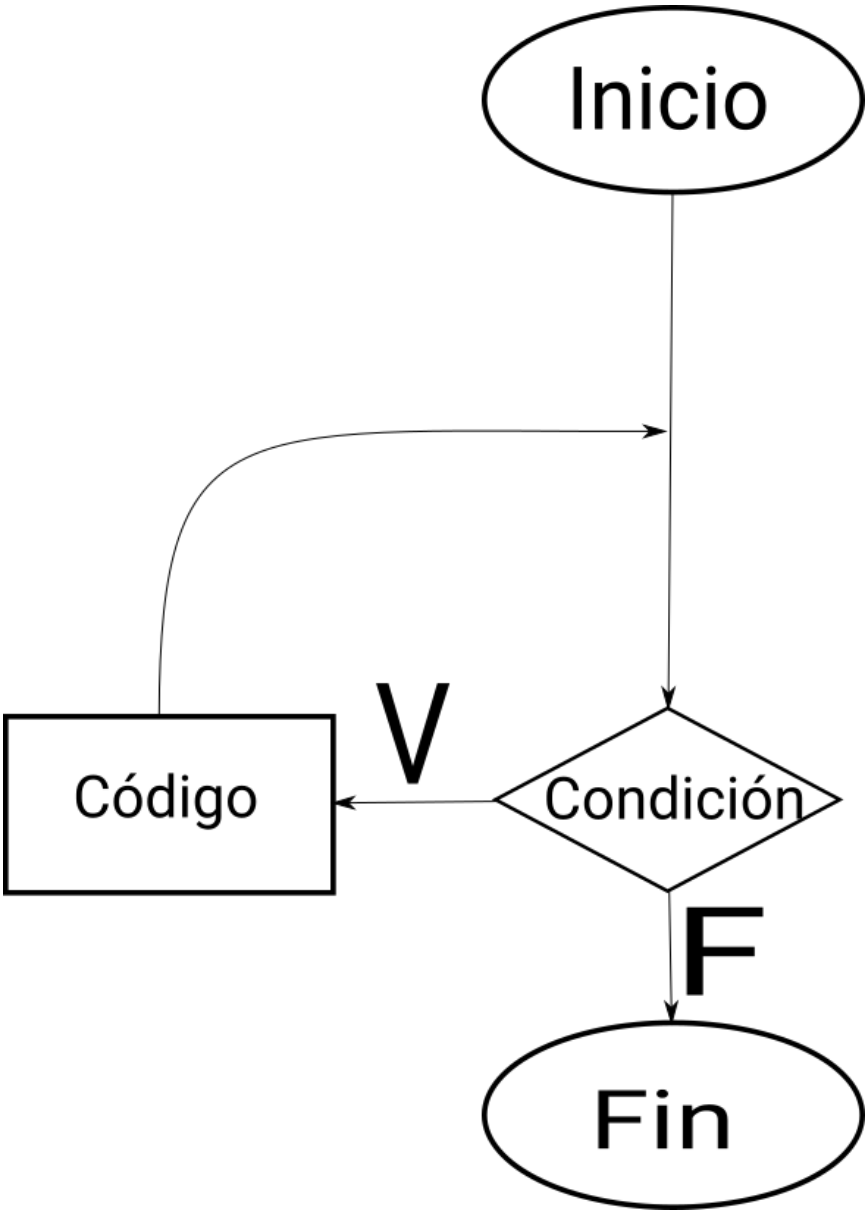

Figure 13: Diagrama de un bucle while

```
    //Si no lo hacemos, el valor de númeroActual siempre será menor o igual a 10 y ten
    numeroActual++;
}
    }

}
```

Figure 14: Bucle While

5.2 El bucle for

El bucle `for` es un caso especial del bucle `while`, este bucle se va a ejecutar igualmente mientras se cumpla la condición dada, por lo que su diagrama es el mismo, pero nos permite simplificar la programación metiendo en la cabecera tanto la inicialización de la variable como su modificación, por ejemplo, si como en el caso anterior queremos escribir los números del 1 al 10 con un bucle `for` lo haríamos así:

```
class BucleFor {

    public static void main (String[] args) {
for (int numeroActual=1; numeroActual<=10; numeroActual++) {
    System.out.println(numeroActual);
}
    }

}
```

Figure 15: Bucle For

Como podemos ver, el resultado de este programa será exáctamente el mismo que el anterior:

```
1
2
3
4
5
6
7
```

```
8
9
10
```

La decisión de utiliar un tipo de bucle u otro depende del programador, pero se suele utiliar el bucle `for` para situaciones en las que haya que *contar*, como en el caso que hemos puesto porque nos permite crear y deshechar la variable en la propia cabecera sin tener que llevar datos innecesarios, aunque por supuesto podemos usar una variable que tengamos de antes como en el bucle `while` e, incluso, no modificar la variable en la cabecera y hacerlo en el cuerpo.

```
class BucleForSinInicializacion {

    public static void main (String[] args) {
    int numeroActual=1;
    for (; numeroActual<=10; numeroActual++) {
    System.out.println(numeroActual);
}
    }

}
```

Figure 16: Bucle For sin inicialización en la cabecera

```
class BucleForSinModificacion {

    public static void main (String[] args) {
for (int numeroActual=1; numeroActual<=10;) {
    System.out.println(numeroActual);
    numeroActual++;
}
    }

}
```

Figure 17: Bucle For Sin Modificacion en la cabecera

Y, por supuesto, si sacamos de la cabecera tanto la inicialización como la modificación de la variable, lo que tenemos es un bucle `while` con otro nombre:

```
class BucleForSinInicializacionNiModificacion {

    public static void main (String[] args) {
int numeroActual=1;
for (; numeroActual<=10;) {
    System.out.println(numeroActual);
    numeroActual++;
}
    }

}
```

Figure 18: Bucle For Sin inicialización ni Modificacion en la cabecera

Existe un último caso de bucle `for`, se trata del también llamado bucle `for-each`, ya que sólo puede ser usado por colecciónes de elementos (no entremos en detalle ahora), y se ejecutará una vez por cada elemento de la colección, lo explicaremos en detalle más adelante pero, suponiendo una colección de elementos de tipo `String` llamada nombres su pinta sería la siguiente:

```
for (String nombre : nombres) {
  System.out.println(nombre);
}
```

Figure 19: Bucle *for-each*

Donde `nombre` es el nombre de la variable que vamos a usar y que va a tomar todos los valores de nuestra colección de `nombres` de uno en uno.

5.3 El bucle do-while

Hasta ahora hemos visto bucles que se ejecutan sólo si se cumple una determinada condición, pero ¿y si queremos que un fragmento de código se ejecute como mínimo una vez pero si se cumple la condición se ejecute unas cuantas mas? Podríamos duplicar el mismo código, una vez fuera del bucle y otra vez dentro, pero para ahorrarnos la reduncancia tenemos el bucle `do-while`.

Este bucle se trata de un bucle `while` en el que la condición para volverlo a ejecutar se encuentra al final y no al principio, fijémonos en el siguiente diagrama:

Podemos continuar con nuestro ejemplo de contar de 1 a 10, veamos como se haría con un bucle do-while:

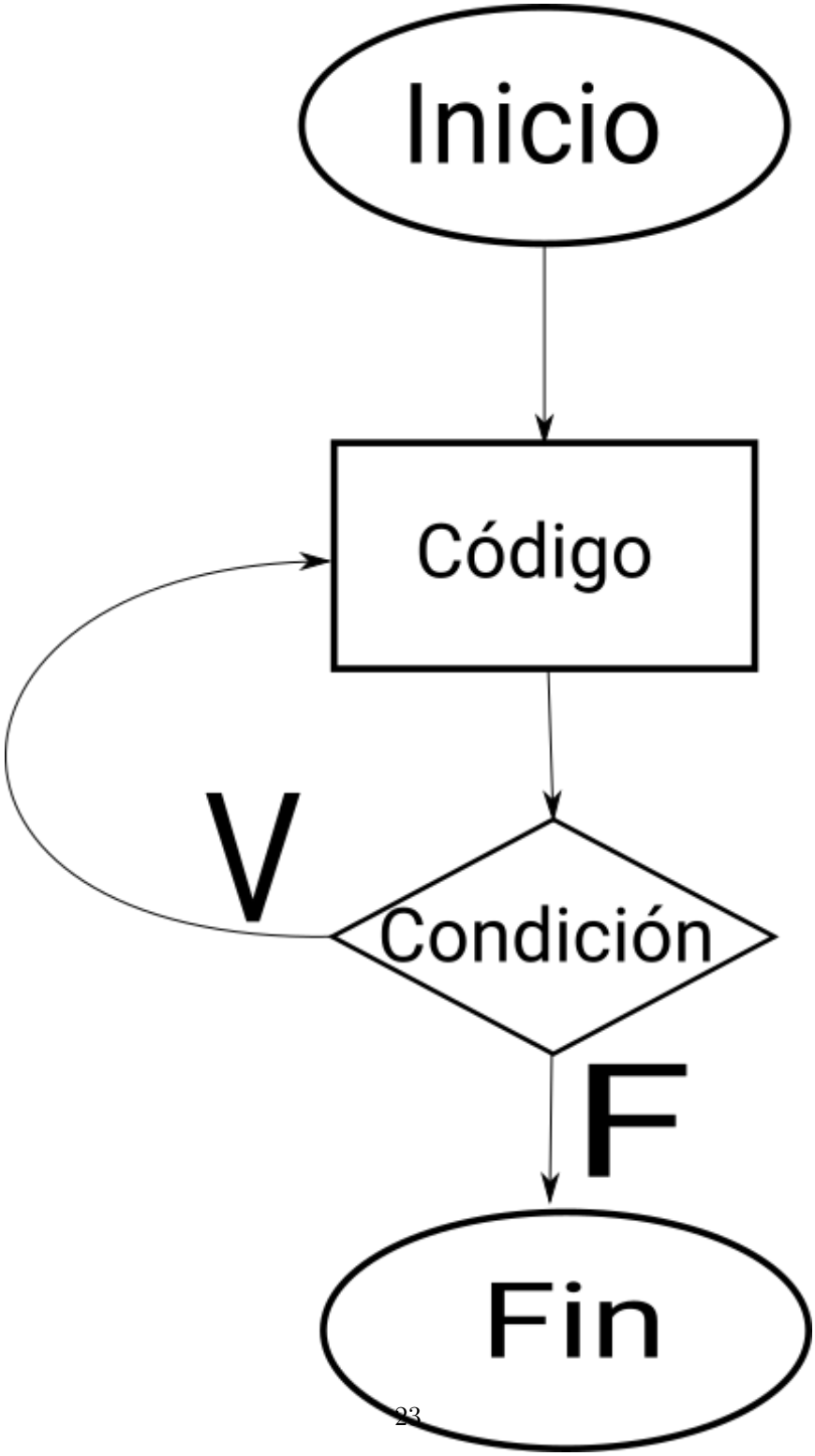

Figure 20: Diagrama de un bucle do-while

```
class DoWhile {

  public static void main(String[] args) {
    //Inicializamos la variable fuera del bucle
    int numeroActual=1;
    do {
      //Imprimimos su valor
      System.out.println(numeroActual);
      //Aumentamos la variable
      numeroActual++;
      //Comprobamos la condición, como ya se ejecuta una vez como mínimo
      //es necesario poner menor (<) y no menor o igual (<=), si lo hiciéramos
      //cuando el valor de numeroActual fuera 10 cumpliría y volvería a ejecutar
      //el código, por lo tanto contaría hasta 11
    } while (numeroActual < 10);
  }

}
```

Figure 21: Bucle do-while

```
1
2
3
4
5
6
7
8
9
10
```

6 Alcance de los elementos

En esta sección hablaremos del ancance de los elementos, puede que programando te hayas dado cuenta de que a veces una variable que has declarado anteriormente no existe cuando la vas a utilizar más tarde, esto es porque una variable sólo está disponible desde el punto en el que se declara, si, pero además desde el bloque en el que se declara, si declaramos, por ejemplo, una

variable dentro de un bloque `if`, no estará disponible fuera de este, si que podríamos declararla antes, inicializarla en el bloque `if` y entonces si estaría disponible y con el valor que le hayamos dado, el bloque más externo al que puede pertenecer una variable es el de la **clase**.

Pero existe una serie de palabras reservadas: `public`, `protected` y `private` que nos servirán para determinar como de aislada estará una determinada variable (o método, o incluso clase) en relación al resto de las clases de un programa.

Ahora hablaremos un poco de lo que significa cada una de estas palabras, no te preocupes si ahora no entiendes que quiere decir cada uno de los términos que usemos, más adelante nos ocuparemos de ello:

- `public`: Indica que el elemento al que se aplique estará disponible para todas las clases del programa y se puede acceder a él sin restricciones.
- `protected`: Indica que el elemento sólo estará disponible para instancias de la clase en la que se encuentre y para clases que hereden de esta (no nos preocupemos por esta terminología ahora)
- `private`: La variable es privada y sólo estará disponible dentro de la propia clase donde se declare.

Hay una palabra modificadora más `static` que se puede aplicar junto las demás (normalmente con `public`), esta palabra reservada nos indica que el elemento al que haga referencia estará disponible incluso sin instanciar un objeto de la clase que lo contiene (Cada vez que usas **System.out.println()**, **System** es una clase, **out** un parámetro de la misma que está declarado como `static` y **println()** un método del objeto out)

7 Manejo de Excepciones

Inevitablemente se producirán errores en nuestro código, es un hecho, muchas veces estos errores serán *en tiempo de compilación*, es decir, se producirán al tratar de compilar el código, aquí encontraremos errores de sintaxis, errores de tipos incompatibles (como el visto en el ejercicio Valor Muy Grande) y cosas así, sin embargo, hay otro tipo de errores llamados de *tiempo de ejecución* que solo detectaremos una vez se esté ejecutando este código, estos errores producirán que nuestro programa termine su ejecución inesperadamente y no cumpla su función, estos errores inesperados se llaman **Excepciones**.

Java lanza una **Excepción** cada vez que se produce un error de este tipo, hay excepciones de muchísimos tipos, sin entrar en detalle todavía, diremos que cada excepción es una **extensión** (una especialización) la clase `Exception`.

Veamos un ejemplo donde se lanzará una excepción:

```
class Excepciones {
  public static void main (String [] args) {
    //Creamos un objeto, un array de Strings en este caso
    String [] objeto = null;

    //En este momento, si intentamos obtener un valor de
    //este objeto, java nos dará un tipo de excepción muy concreto
    //indicando que el objeto es nulo
    objeto.toString();
  }
}
```

Figure 22: Excepciones

```
Exception in thread "main" java.lang.NullPointerException
        at Excepciones.main(Excepciones.java:9)
```

7.1 Los bloques try-catch

Afortunadamente Java nos proporciona una funcionalidad para capturar y controlar estos errores, se trata de los bloques *try-catch*.

Los bloques *try-catch* se tratan de bloques que empiezan con una sentencia `try` y dentro de ella, un bloque donde preveemos que se puede producir una excepción, seguida por uno o más bloques `catch`, cada uno de ellos encargado de *capturar* una posible excepción.

Veamos un ejemplo con el anterior bloque de código:

```
class TryCatch {
  public static void main (String [] args) {
    //Creamos un objeto, un array de Strings en este caso
    String [] objeto = null;

    //Abrimos un bloque try, porque sabemos que aquí se puede
    //producir una excepción
```

```
    try {
      //En este momento, si intentamos obtener un valor de
      //este objeto, java nos dará un tipo de excepción muy concreto
      //indicando que el objeto es nulo
      objeto.toString();
    } catch(NullPointerException npe) {
      //En el bloque catch, capturamos la excepción, en este caso
      //una NullPointerException, a la que llamamos npe y que
      //dentro de este bloque podemos usar como variable.
      System.err.println("Se ha producido una NullPointerException");
    }
  }
}
```

Figure 23: Try-Catch

```
Se ha producido una NullPointerException
```

Si modificamos un poco el código anterior podemos ilustrar el hecho de que se pueden poner varios bloques `catch` en una sola sentencia `try`, a lo que añadiremos uno más, el bloque `finally`, que se ejecutará siempre, independientemente de que se produzca una excepción (y se entre en el bloque catch) o no se produzca (y, por lo tanto, no se entre en el bloque catch)

```
class TryCatchMultipleFinally {
  public static void main(String[] args) {
    // Creamos un objeto, un array de Strings en este caso
    String[] objeto = null;

    // Abrimos un bloque try, porque sabemos que aquí se puede
    // producir una excepción
    try {
      // En este momento, si intentamos obtener un valor de
      // este objeto, java nos dará un tipo de excepción muy concreto
      // indicando que el objeto es nulo
      objeto.toString();
    } catch (NullPointerException npe) {
      // En el bloque catch, capturamos la excepción, en este caso
      // una NullPointerException, a la que llamamos npe y que
      // dentro de este bloque podemos usar como variable.
```

```
  System.err.println("Se ha producido una NullPointerException");
} catch (IndexOutOfBoundsException ioobe) {
  System.err.println("Se ha producido una IndexOutofBoundsException");
} catch (Exception e) {
  System.err.println("Se ha producido otro tipo de Excepción");
} finally {
  System.out.println("Bloque finally");
}

// Inicializamos el objeto sin nada dentro
objeto = new String[] {};

try {
  // Intentamos obtener el primer valor del objeto
  System.out.println(objeto[0]);
} catch (NullPointerException npe) {
  System.err.println("Se ha producido una NullPointerException");
} catch (IndexOutOfBoundsException ioobe) {
  System.err.println("Se ha producido una IndexOutofBoundsException");
} catch (Exception e) {
  System.err.println("Se ha producido otro tipo de Excepción");
} finally {
  System.out.println("Bloque finally");
}

// Inicializamos el objeto con un valor
objeto = new String[] {"Primer Valor"};

try {
  // Intentamos obtener el primer valor del objeto
  System.out.println(objeto[0]);
} catch (NullPointerException npe) {
  System.err.println("Se ha producido una NullPointerException");
} catch (IndexOutOfBoundsException ioobe) {
  System.err.println("Se ha producido una IndexOutofBoundsException");
} catch (Exception e) {
  System.err.println("Se ha producido otro tipo de Excepción");
} finally {
  System.out.println("Bloque finally");
}
```

```
  }
}
```

Figure 24: Try-Catch Múltiple y Finally

```
Se ha producido una NullPointerException
Bloque finally
Se ha producido una IndexOutofBoundsException
Bloque finally
Primer Valor
Bloque finally
```

8 Métodos y funciones

Con lo que ya sabemos podemos crear programas muy potentes, podemos controlar si un bloque de código se ejecutará o no y cuantas veces lo hará, pero la ejecución sigue siendo *de arriba a abajo*, estamos en lo que se conoce como **programación estructurada**, pero vayamos más allá, hasta ahora, si queríamos ejecutar un bloque de código más de una vez podíamos hacer bucles, pero estos siempre se ejecutarán con los mismos datos, la estructura de un método es la siguiente:

```
public static void main (String [] args)
```

Figure 25: Estructura de un método

- En azul vemos los modificadores, no nos pararemos ahora a explicarlos en detalle, digamos que es un método público (que puede ser accedido desde cualquier clase) y estático (que puede ser accedido sin necesidad de crear una instancia del método)
- En rojo tenemos el *tipo de retorno*, puede ser cualquiera de los tipos de dato conocidos (int, char, long, boolean...), cualquier tipo de clase (como por ejemplo, String), o `void`, que significa que el método no devolverá nada.
- En morado vemos el nombre del método, es el que utilizaremos para invocarlo

- A continuación tenemos los paréntesis, en estos paréntesis irán los parámetros que entrarán al método, declarados de la misma manera que se hace con las variables (tipo de variable seguido del nombre de la variable), separados por comas, o bien nada, con el paréntesis vacío si el método no necesita parámetros.

Hay un tipo de paso de parámetros un poco especial, conocido como *varargs*, que consiste en que un método recibirá un número indeterminado de parámetros de un tipo, se escribe poniendo tres puntos detrás del tipo de variable, antes del espacio y el nombre y solo puede ser usado una vez por método y al final de la lista de parámetros. Como ilustración, este tipo de parámetro se escribe de la siguiente manera:

```
public String varargs(String... parametros)
```

Figure 26: Varargs

Esa variable `parametros` que tenemos en el ejemplo anterior, se accederá como si fuera un objeto de tipo array del tipo que se indique (en este caso String).

Veamos ahora un ejemplo sencillo de la potencia de un método con el que podemos ejecutar un mismo código con **parámetros** diferentes:

```
class MiPrimerMetodo {

  //Aquí tenemos el método main, no devuelve nada (void)
  //y recibe como parámetro un array de Strings (varias cadenas de texto)
  public static void main(String [] args) {
    //Creamos una variable llamada nombre y la inicializamos
    String nombre = "Bimo";
    //Llamamos a nuestro método y le pasamos como parámetro la variable
    saludar(nombre);
    //Cambiamos el valor de la variable
    nombre = "Kirby";
    //Y volvemos a llamar al método
    saludar(kirby);
  }

  //Aquí tenemos nuestro método, se llama saludar y no devuelve nada,
  //recibe como parámetro una cadena de texto llamada nombre,
  //cada vez que se ejecute saludará a quien venga escrito en la variable nombre
```

```
  static void saludar(String nombre) {
    System.out.println("Hola, "+nombre);
  }

}
```

Figure 27: Mi primer método

```
Hola, Bimo
Hola, Kirby
```

8.1 Paso de parámetros

En el ejemplo anterior hemos visto que al método *saludar* le pasábamos un parámetro, un nombre en este caso, de tipo String, cuando declaramos un método, dentro de los paréntesis le podemos poner, separados por comas, todos los parámetros que vaya a necesitar para cumplir su función, estos parámetros se escriben de la misma manera que se creaban las variables (pero sin inicializar), y cuando se les invoca se le pasan las variables necesarias que se correspondan con los parámetros que requiere, veamos un ejemplo:

```
//Supongamos que llegamos a este método,
//como se ve, no tiene parámetros ni devuleve nada (void)
public void metodo1() {
  System.out.println("El número es: "+metodo2());
}

//Este método es privado, sólo se puede invocar desde
//esta clase, y devuelve un número entero,
//tampoco recibe parámetros
private int metodo2() {
  int numero = metodo3(2);
  return numero;
}

//Este es otro método privado, que recibe como
//parámetro un número entero y llama al metodo4
private int metodo3(int numero) {
  return metodo4(numero, 0.5);
}
```

```
//Este método recibe como parámetros un número entero
//y un número decimal y los multiplica, después devuleve
//el resultado entero.
private int metodo4(int numero, double factor) {
  return (Integer)(numero * factor);
}
```

En Java, se dice que los parámetros que se pasan a los métodos se hacen *por valor*, esto quiere decir que cada método hace una copia del parámetro cuando se invoca y, se asignamos un nuevo valor al parámetro dentro del método, este no cambiará en el método que lo invocó, veamos un ejemplo:

```
class PasoDeValor {

  //Declaramos un método al que llamaremos,
  //Como podemos ver, podemos declararlo antes
  //del método main, aunque se le llamará después
  static void cambiarValor(int numero) {
    numero = 2;
  }

  public static void main (String [] args) {
    //Declaramos una variable y le asignamos un valor
    int numero = 1;

    //Llamamos al métod de cambio de valor
    cambiarValor(numero);

    //Mostramos el resultado por pantalla
    System.out.println(numero);
  }

}
```

Figure 28: Paso de valor

```
1
```

Se puede pensar que el que los valores no pasen de los métodos a quien los llamó es poco util, pero si puede hacerse, para ello usaremos la sentencia

return, que literalmente *devuelve* el valor que le digamos, si modificamos un poco el programa anterior lo veremos:

```
class RetornoDeValor {

  //Declaramos un método al que llamaremos,
  //Como podemos ver, podemos declararlo antes
  //del método main, aunque se le llamará después.
  //En este caso hemos cambiado el tipo de retorno del método
  // de void (no devuleve nada) a int, y hemos añadido la sentencia
  // return con nuestra variable.
  static int cambiarValor(int numero) {
    numero = 2;
    return numero;
  }

  public static void main (String [] args) {
    //Declaramos una variable y le asignamos un valor
    int numero = 1;

    //Llamamos al métod de cambio de valor
    //y le asignamos el valor del retorno
    numero = cambiarValor(numero);

    //Mostramos el resultado por pantalla
    System.out.println(numero);
  }

}
```

Figure 29: Retorno de valor

```
2
```

Ahora, para ver otra característica del paso de parámetros a los métodos, tenemos que hacerlo mediante objetos, de momento no nos preocupemos mucho de ello, básicamente lo que tenemos que tener en cuenta es que si pasamos un objeto como parámetro y cambiamos el valor de una de sus propiedades dentro de un método, esta permanecerá cambiada incluso fuera del método, esto puede parecer lo opuesto a lo que acabamos de ver, pero no

es así, si en lugar de cambiar el valor de una de las propiedades del objeto lo que hiciéramos fuera instanciar un nuevo objeto en la variable (como asignar un nuevo valor en las variables que ya conocemos), el valor de la variable original permanecería intacto, veamoslo de nuevo con dos ejemplos:

```
import java.util.ArrayList;

class CambioDeObjeto {
  public static void main (String [] args) {
    /*
      Creamos un objeto de tipo ArrayList, este objeto consiste un una lista del tipo d
      que le digamos entre los acentos, en este caso Integer (número entero), se declar
      sabemos TipoDeObjeto nombreDeLaVariable, y luego se le asigna un valor, en este c
      nuevo de la clase ArrayList, los paréntesis que aparecen al final son porque esta
      al método constructor del objeto, sin parámetros en este caso, aunque puede haber
     */
    ArrayList<Integer> lista = new ArrayList<Integer>();

    /*
      Añadimos un objeto de la clase Integer (un 1), en el objeto lista, para ello llam
      add de la clase ArrayList, con un parámetro de tipo entero (el 1), este método aí
     */
    lista.add(1);

    //Imprimimos el valor del primer elemento de la lista (posición 0), llamando al mét
    System.out.println(lista.get(0));
  }

  /*
    Se asigna un nuevo valor a la variable lista
   */
  static void nuevoObjeto(ArrayList<Integer> lista) {
    lista = new ArrayList<Integer>();
    lista.add(2);
  }
}
```

Figure 30: Cambio de objeto

```
1
```

Ahora vamos a ver como podemos añadir un valor al objeto que ya existe

```
import java.util.ArrayList;

class CambioDePropiedad {
  public static void main (String [] args) {
    /*
      Creamos un objeto de tipo ArrayList, este objeto consiste un una lista del tipo d
      que le digamos entre los acentos, en este caso Integer (número entero), se declar
      sabemos TipoDeObjeto nombreDeLaVariable, y luego se le asigna un valor, en este c
      nuevo de la clase ArrayList, los paréntesis que aparecen al final son porque esta
      al método constructor del objeto, sin parámetros en este caso, aunque puede haber
     */
    ArrayList<Integer> lista = new ArrayList<Integer>();

    //Llamamos al método para asignar un nuevo valor a la propiedad
    anadirValor(lista);

    /*
      Añadimos un objeto de la clase Integer (un 1), en el objeto lista, para ello llam
      add de la clase ArrayList, con un parámetro de tipo entero (el 1), este método aí
      En este caso, se añadirá en la posición 1 (la segunda).
     */
    lista.add(1);

    //Imprimimos el valor del primer elemento de la lista (posición 0), llamando al mét
    System.out.println(lista.get(0));
  }

  /*
    Se asigna un nuevo valor a la variable lista
   */
  static void anadirValor(ArrayList<Integer> lista) {
    lista.add(2);
  }
}
```

Figure 31: Cambio de propiedad

```
2
```

8.2 Sobrecarga de métodos

Expliquemos ahora un nuevo concepto, se trata de la *sobrecarga de métodos*, una función extremádamente útil que tiene Java, a pesar del nombre tan rimbombante que tiene se trata simplemente de la posibilidad de tener un método con el mismo nombre pero con diferentes parámetros, sean o bien diferente número o diferente tipo (o ambas, obviamente).

La forma que tiene Java de permitir esto es asignando a cada método una *firma*. La firma de un método viene dada por su nombre y los tipos de sus atributos, es importante porque en una misma clase **no puede haber dos o más métodos con la misma firma**. Veamos un ejemplo:

```
class SobrecargaDeMetodos {

  public static void main (String[] args) {

    saludar("Víctor");
    saludar("Víctor","Toledo");
    saludar(30);
    saludar("Víctor",30);
    saludar("Víctor","Toledo",30);
    saludar();

  }

  static void saludar(String nombre) {
    System.out.println("Hola, "+nombre+". No se tu edad ni donde naciste :(");
  }

  static void saludar(String nombre, String ciudad) {
    System.out.println("Hola, "+nombre+", no se tu edad, pero se que naciste en "+ciuda
  }

  static void saludar(int edad) {
    System.out.println("Hola, no se como te llamas ni donde naciste, pero se que tienes
  }

  static void saludar(String nombre, int edad){
    System.out.println("Hola, "+nombre+", naciste hace "+edad+" pero no se dónde");
  }
```

```
  static void saludar(String nombre, String ciudad, int edad) {
    System.out.println("Hola "+nombre+"! Te conozco bien, tienes "+edad+" años y nacist
  }

  static void saludar() {
    System.out.println("Hola, no te conozco en absoluto, no se ni tu edad, ni dónde nac
  }

}
```

Figure 32: Sobrecarga de métodos

```
Hola, Víctor. No se tu edad ni donde naciste :(
Hola, Víctor, no se tu edad, pero se que naciste en Toledo
Hola, no se como te llamas ni donde naciste, pero se que tienes 30 años
Hola, Víctor, naciste hace 30 años pero no se dónde
Hola Víctor! Te conozco bien, tienes 30 años y naciste en Toledo
Hola, no te conozco en absoluto, no se ni tu edad ni dónde naciste :'(
```

Fijémonos ahora en las firmas de las diferentes sobrecargas del método *saludar*

1. saludar(String)
2. saludar(String, String)
3. saludar(int)
4. saludar(String, int)
5. saludar(String, String, int)
6. saludar()

Podemos ver que en las firmas a Java *le da igual* el nombre de los parámetros, solo le importa su tipo, esto nos limita a que, por ejemplo no podríamos tener un método con que nos saludase con la ciudad de nacimiento y la edad de la siguiente forma:

```
static void saludar(String ciudad, int edad){
```

Ya que tendría la misma firma que el método **4** *saludar(String, int)*, no obstante si que podríamos haberlo añadido de la siguiente manera:

```
static void saludar(int edad, String ciudad){
```

Ya que aquí el método tendría una firma que sería *saludar(int, String)*, que no coincide con ninguna de las otras 6.

8.3 Recursividad

Otra de las cosas a las que no da acceso un método es el concepto de la *recursividad*, esto no es más que un método que se llama a si mismo, por lo tanto tendremos que parar esas llamadas cuando se cumpla una determinada condición como ocurría con los bucles o tendremos un bucle de recursividad infinito y nuestro programa no avanzará.

Un ejémplo clásico de recursividad es el cálculo de un factorial, pero ántes veamos como sería diseñar el cálculo de un factorial con un bucle.

El factorial de un número se define como la multiplicación de si mismo con todos sus números anteriores. Por ejemplo el factorial de 6 ó **6!** = 6*5*4*3*2*1, una de las principales razones de que el factorial se use como ejemplo para la recursividad en programación es que el factorial ES recursivo, si vemos, por ejemplo el factorial de 5 (**5!**) = 5*4*3*2*1, si se compara con 6! como hemos visto antes vemos un patrón, y es que 6! = 6*5!, y así sucesivamente.

```
private static long factorialLoop (int n) {
    long resultado = 1;
    while (n>1) {
resultado *= n--;
    }
    return resultado;
}
```

Figure 33: Factorial con bucle

El método anterior recibe un número entero y si es mayor que 1 va multiplicando el resultado por el número `n` y reduce su valor en 1 hasta que es 1 y sale del bucle.

A continuación el mismo método utilizando recursividad, se puede observar que es bastante similar, pero conceptualmente es más coherente con la definición del factorial:

```
private static long factorialRecur (int n) {
```

```
    long resultado=1;
    if (n > 1) {
       resultado = n*factorialRecur (n-1);
    }
    return resultado;
}
```

Figure 34: Factorial con recursividad

En este caso se entra al método y si n es mayor que 1 se multiplica su valor por el resultado de llamarse a si mismo decrementando en 1 el valor de n. Conviene saber aquí que java, ante una operación da preferencia a la llamada de un método, por ejemplo, la traza de llamar al método `factorialRecur(4)` sería algo similar a lo siguiente:

entramos en factorialRecur(4) -> entramos en factorialRecur(3) -> entramos en factorialRecur(2) -> entramos en factorialRecur(1) -> factorialRecur(1) devuelve 1 -> factorialRecur(2) multiplica 2*1 y devuelve 2 -> factorialRecur(3) multiplica 3*2 y devuelve 6 -> factorialRecur(4) multiplica 4*6 y devuelve 24

Es decir, el método recursivo va *bajando* por las llamadas hasta llegar a lo que se llama **caso base**, en este caso el factorial de 1, que es 1 (aunque lo correcto hubiera sido llegar hasta 0!, que también es 1, pero el resultado es el mismo) y después, según los métodos van llegando a su sentencia `return` vuelve a *subir* hasta el caso de la primera llamada (factorialRecur(4) en nuestro caso).

9 Clases y objetos

Como ya hemos comentado anteriormente, Java es un lenguaje de programación *orientado a objetos*, en Java, prácticamente todo es un objeto, salvo los tipos de datos primitivos (int, char, boolean,...) que incluso también tienen una clase asociada (Integer, Character y Boolean, respectivamente para los ejemplos anteriores, nótese que empiezan por mayúscula), aunque probablemente, la clase con la que estarás más familiarizada será la clase String, usada para cadenas de texto.

Muchas veces se usan indistintamente los términos clase y objeto, lo correcto es llamar clase al archivo fuente .java y objeto a las *instancias* de este.

9.1 El método constructor

En Java (y en cualquier otro lenguaje de programación orientado a objetos), llamamos instanciar al hecho de crear un objeto de una determinada clase en nuestro código, para ello hacemos uso de un tipo de método especial llamado **constructor**.

Este método es especial por varios motivos:

1. Es un método sin nombre, su nombre es el nombre de la propia clase
2. Es necesario invocarlo unido a la palabra reservada **new**

Esto lo hemos visto anteriormente en el ejemplo de paso de variables por valor a los métodos de la siguiente manera:

```
ArrayList<Integer> lista = new ArrayList<Integer>();
```

Analizando la sentencia anterior tenemos lo siguiente:

- ArrayList<Integer>: El tipo de objeto que vamos a crear, exáctamente igual que cuando creábamos variables de tipo int, char, boolean, String,.... En este caso, un objeto ArrayList (Un tipo de lista) de objetos de tipo Integer (o int, números enteros)
- lista: El nombre de la variable, nada nuevo aquí.
- new ArrayList<Integer>(): y aquí tenemos la llamada al método constructor, como podemos ver, es un método que se llama igual que la clase que queremos instanciar (ArrayList<Integer>) y que va seguido por unos paréntesis, en este caso, sin nada dentro porque no le pasamos ningún parámetro, todo ello precedido de la palabra reservada **new**, que es la que le indica a Java que lo que venga detrás será un método constructor.

En Java, por convención, las clases empiezan con mayúscula, meintras que los nombres de las variables y los métodos se escriben empezando con minúscula, en ambos casos siguiendo la convención *CamelCase* (*UpperCamelCase*, comenzando con mayúscula para las clases y *lowerCamelCase*, comenzando con minúscula para lo demás), que consiste en, si necesitamos varias palabras en el nombre, separarlas con mayúsculas y sólo usarlas en ese caso o en caso de abreviaturas conocidas, por ejemplo, podríamos tener un método que se llamase *getSQL*, ya que *SQL* es una abreviatura bien conocida.

9.2 El caso de la clase String

Anteriormente hemos instanciado una clase que es un caso especial, no lo hemos hecho mediante un método constructor, sino asignándole un valor con = como hacíamos con los tipos primitivos, se trata de la clase String.

Esta clase, por ser tan extendida se crea de esta manera, que nos puede dar la impresión de que se trata de un tipo de variable primitivo:

```
String nombre = "Kirby";
```

Pero en realidad lo que estábamos haciendo era crear una nueva instancia (un nuevo objeto) de la clase String.

La clase String, como cualquier otra clase en Java necesariamente debe tener al menos un método constructor, y así es, es sólo que Java nos ofrece esta conveniencia.

9.3 El valor null

Hasta ahora, cuando creábamos una variable esta podía estar inicializada o no, y, si lo estaba, tenía un valor válido dentro de su tipo, el valor **null** es un valor especial, nos permite indicar que el contenido de una variable de tipo objeto es nulo, es decir, no contiene nada pero existe como variable, el valor **null** es el mismo para cualquier tipo de objeto, se puede asignar a cualquier objeto con el operador de asignación (=) y puede ser usado en operaciones de comparación, vamos a ver un ejemplo:

```
class ValorNull {
  public static void main(String [] args) {

    //Asignamos el valor null a una objeto instancia de la clase String
    String nombre = null;

    //Comprobamos si la variable nombre es nula (valor null)
    if (nombre == null) {
      nombre = "Kirby";
    }

    System.out.println(nombre);

  }
}
```

Figure 35: Valor null

`Kirby`

9.4 Clases más conocidas

9.4.1 La clase Object

En Java todos los objetos que exiten, ya sean los que nos proporciona el propio lenguaje o los que creemos nosotros extienden de la clase `Object`. Object es la clase primaria del lenguaje de programación Java, no se usa mucho como tal, pero es conveniente conocer que existe, podemos declarar un objeto de esta clase e instanciarlo con cualquier otra, es la madre de todas las demás.

9.4.2 String

Como acabamos de ver, la clase String probablemente sea la clase más conocida de Java, empezamos a trabajar con ella prácticamente en el primer programa HolaMundo que hacemos y la utilizamos tanto como a los tipos primitivos, sirve para guardar cadenas de texto y no nos detendremos mucho más en ella.

1. Métodos

    - equals(objeto): Aunque este método es genérico para todos los objetos, es importante recalcarlo, ya que muchas veces querremos comparar si una cadena de texto es igual a otra.
    - equalsIgnoreCase(objeto): Este método es igual que el anterior salvo que ignora las mayúsculas y las minúsculas.
    - charAt(indice): Devuelve la letra (como tipo `char`) que haya en el índice que le pasemos.
    - length(): Nos devuelve el tamaño del texto.
    - indexOf(cadena): nos devuelve la primera posición en la que aparece la cadena.
    - lastIndexOf(cadena): igual que el anterior, pero nos devuelve la última.
    - toUpperCase(): Convierte la cadena en mayúsculas.
    - toLowerCase(): Convierte la cadena en minúsculas.
    - valueOf(objeto): Método estático que devuelve el valor del objeto en forma de string.

9.4.3 Clases asociadas a tipos primitivos

Como ya vimos en la sección Tipos de datos, los tipos primitivos tienen una clase asociada a cada uno de ellos, el nombre del tipo de dato es normalmente una abreviatura del nombre completo de la clase y se escribe en minúscula, mientras que la clase, como hemos visto antes, se escribirá en mayúscula. Estas clases son especialmente útiles, ya que, además de contener una serie de métodos que nos ofrecen funciones para trabajar con estos tipos de datos, pueden ser usados para almacenar el valor **null** en ellos, indicando así que un dato que normalmente tendría un valor por defecto, no contiene nada.

1. Métodos Uno de los métodos más importantes de estas clases es el método estático parse... (cadena), los puntos suspensivos son porque su nombre es diferente para cada clase, por ejemplo Long.parseLong() para la clase Long, Integer.parseInt() para la clase Integer, etc., este método convierte una cadena en el tipo que haga referencia, es especialmente útil cuando queremos que un usuario nos introduzca un valor por la consola, ya que siempre llegará en forma de String y si lo que necesitamos es un número deberemos convertirlo de esta manera.

9.4.4 Matrices (o Arrays)

Las matrices (o Arrays, como se llaman en inglés y, comunmente en los entornos de trabajo) pertenecen a un grupo de objetos que se conocen como **colecciones**, estas colecciones consisten en una agrupación de varios objetos, en el caso de los arrays, una lista. Los arrays lo pueden ser de cualquier tipo de objeto (pero sólo de uno a la vez) y se identifican porque se declaran con el nombre del tipo del que queremos la lista y unos corchetes `[]`.

Para insertar o recoger los valores de la lista lo haremos llamando a la variable como hacíamos habitualmente y añadiendo un número entre corchetes, como vimos en el ejercicio TryCatchMultipleFinally, en ese ejercicio intentábamos obtener el primer valor (índice 0) de la variable `objeto` y lo hacíamos de la siguiente manera:

```
objeto[0];
```

Esa simple instrucción nos apunta a la primera posición del array, hay que tener en cuenta que los arrays, como la gran mayoría de estructuras múltiples que existen en los lenguajes de programación, se empiezan a contar desde 0 y no desde 1, es decir, en un array con un tamaño de 5, las posiciones a las que podemos acceder son las 0, 1, 2, 3 y 4; si intentásiemos acceder

a la posición 5, Java nos lanzaría una `IndexOutOfBoundsException` como vimos en el ejercicio mencionado anteriormente. Esta excepción nos indica que hemos intentado acceder a un índice (posición) más allá de los límites del objeto (El objeto tiene 5 posiciones y hemos intentado acceder a la 6ª).

Una vez tenemos una posición que existe, podemos asignarle un valor o recogerlo como hacíamos con cualquier otro tipo de variable. Para ilustrarlo:

Igual que hacíamos:

```
class AccesoVariable {
  public static void main (String [] args) {
    int numero; //Declaramos la variable
    numero = 2; //inicilizamos la variable
    System.out.println(numero); //Obtenemos el valor de la variable
  }
}
```

Figure 36: Acceso a variable

```
2
```

Podemos hacer:

```
class AccesoArray {
  public static void main (String [] args) {
    int[] array; //Declaramos la variable
    array = new int[]{2}; //Inizializamos la variable (con un valor en la posición 1)
    System.out.println(array[0]); //Obtenemos el valor de la posición 1
  }
}
```

Figure 37: Acceso a posición de array

```
2
```

1. Declaración e inicialización Como todos los objetos, los arrays pueden ser inicializados con la palabra reservada `new` y llamando al constructor, pero esta clase, al igual que la clase `String` es una clase especial y puede ser inicializada de varias maneras.

(a) Inicialización con constructor vacío En este caso lo haremos de la siguiente manera, por ejemplo, para un array de números enteros (int o Integer):

```
int[] miArray = new int[5];
```

Figure 38: Inicialización de un array con constructor

Observamos que también se trata de un caso especial (no hay paréntesis en la llamada al método constructor), de izquierda a derecha, lo que tenemos es:

- Antes del =:
  - int[]: Tipo de dato, array ([]) de números enteros (int).
  - miArray: El nombre de la variable.
- Después del =:
  - new: La palabra reservada que nos indica que lo que queremos es invocar al método constructor.
  - int[5]: La llamada propiamente dicha, el número que vemos entre los corchetes será el tamaño de la lista, hemos de tener en cuenta que este tamaño es fijo, por lo que no nos conviene quedarnos cortos, pero sería un desperdicio de memoria si nos pasamos, es un valor que hay que pensar muy bien.

Ahora ya podríamos añadir valores a nuestro array, por ejemplo, añadamos un 2 en la posición 4:

```
miArray[3] = 2;
```

Figure 39: Añadir valor a array

(b) Inicialización con constructor con datos El caso anterior, aunque util de conocer, en ocasiones nos parecerá engorroso, pues tendremos que rellenar el array escribiendo una sentencia para cada posición que tenga, afortunadamente, en Java se puede inicializar el array directamente en el constructor de la siguiente manera:

```
int[] miArray = new int[]{5,4,3,2,1};
```

Figure 40: Array con datos

De esta manera no nos hace falta indicarle al constructor cuantos elementos tendrá el array, en cambio, le decimos los valores en

orden que almacenará (en nuestro caso; 5 en la posción 1, 4 en la posición 2, 3 en la posición 3, etc.)

(c) Inicialización con datos sin constructor Este es el caso más especial, y es que podemos evitar escribir la parte del constructor y poner solo las llaves con los datos si lo hacemos justo cuando declaremos la variable:

```
int[] miArray = {5,4,3,2,1};
```

Figure 41: Array con datos sin constructor

Si lo quisiéramos separar de la declaración, este formato daría error, ya que Java no tiene manera de asegurarse de qué tipo es lo que queremos asignar a la variable, en este caso habría que usar el constructor anterior:

```
int[] miArray;

miArray = {5,4,3,2,1}; //Esto da error

miArray = new int[]{5,4,3,2,1}; //Esto funciona
```

Figure 42: Declaración separada

2. Arrays múltiples Otra cosa interesante que nos permiten hacer los arrays es añadir unos dentro de otros, obteniendo así estructuras *multidimensionales*.

   Si entendemos que un array `[]` es una lista (o una línea de valores, si queremos), si cada elemento del array es otro array `[][]` lo que tenemos es un cuadro y, si a su vez, cada elemento del cuadro es otra matriz `[][][]` lo que tenemos es un cubo de valores y, para acceder a cada uno de ellos es necesario indicar cada una de las coordenadas que indican su posición.

   Los arrays multidimensionales se comportan exáctamente igual que los planos, sólo hay que tener en cuenta las dimensiones extra para tener claro a qué valor estamos accediendo, que para inicializarlos es necesario indicar el tamaño de cada uno de las dimensiones (no tienen por qué ser regulares) y que si queremos inicializarlo directamente con valores, cada elemento que va entre las comas dentro de las llaves serán a

su vez otras llaves hasta que se haya profundizado en cuantas dimensiones tenga, por ejemplo, un array multidimensional lo podríamos instanciar así:

```
int[][] miArrayMultidimensional = new int[3][12];

String[][] otroArrayMultidimendional = {{"p11","p12","p13"},{"p21","p22","p23"}};
```

Figure 43: Array Multidimensional

3. Métodos y propiedades
   - lentgh: propiedad que nos indica la longitud del array
4. Recorriendo un Array Dada su naturaleza, muchas veces nos interesará recorrer todos los valores de un array, ya sea para buscar un valor concreto si no sabemos su posición, para imprimir todos los valores o para cualquier otra cosa que se nos pueda ocurrir. Esto se puede conseguir de múltiples maneras, pero las más común sería utilizando un bucle **for**.

   Por ejemplo, si tenemos el siguiente Array de Strings

   ```
   String[] nombres = {"Pepe","Juan","María","Beatriz","Víctor","Raquel"};
   ```

   Podríamos hacer un bucle for clásico para recorrer e imprimir cada uno de los valores del array:

   ```
   //Nuestro bucle for recorrerá desde la posición 0, hasta la última del array
   //identificada por su propiedad length y sumará 1 a la variable i en cada
   //iteración del bucle
   for (int i=0; i<nombres.length; i++) {
     System.out.println("Hola, "+nombres[i]);
   }
   ```

   Pero como vimos en la sección dedicada al bucle for, teníamos un tipo de bucle que se ejecuta una vez por cada elemento de una colección y, efectivamente, un array es una colección de elementos como habrás podido deducir. Por lo tanto, usando esta manera simplificaremos el bucle:

```
for (String nombre : nombres) {
  // Ya no tenemos una variable i con la que acceder a cada posición del array nom
  // sino que tenemos una variable nombre, del mismo tipo que cada elemento del ar
  // de tipo String, que en cada iteración contendrá el un valor del array de nomb
  System.out.println("Hola, "+nombre);
}
```

En ambos casos el resultado será el mismo:

```
Hola, Pepe
Hola, Juan
Hola, María
Hola, Beatriz
Hola, Víctor
Hola, Raquel
```

9.4.5 Date

La clase Date del paquete java.util es una clase que nos permite almacenar fechas.

1. Declaración e inicialización Como cualquier objeto, la clase Date se inicializa a través de la palabra reservada `new`:

   `Date` tiene dos métodos constructores (tiene más, pero están en desuso), el primero no admite parámetros y nos muestra la fecha actual:

```
import java.util.Date;

class FechaActual {

  public static void main (String [] args) {
    Date fechaActual = new Date();
    System.out.println(fechaActual);
  }
}
```

Figure 44: Constructor Date con fecha actual

```
Sun May 31 00:30:25 CEST 2020
```

El otro método constructor del que disponemos nos permitirá crear un objeto Date con la fecha que le digamos, pero su formato es un poco extraño, ya que requiere que le pasemos el número de milisegundos que han pasado desde una fecha que se toma como referencia (conocida como Epoch) y que es el 1-1-1970, como no nos vamos a poner a calcularlos, podemos utilizar una de la múltiples webs que nos ofrecen esta información en internet, aqui utilizaremos Epoch converter - Unix Timestamp converter, pongamos por ejemplo el 2-2-1990, según la web, el número de milisegundos que pasaron desde Epoch hasta esa fecha es 633969132000, veamos si coincide:

```
import java.util.Date;

class OtraFecha {

  public static void main (String [] args) {
    //Como el parámetro es de tipo long tiene que llevar una L al final para disti
    Date otraFecha = new Date(633969132000L);
    System.out.println(otraFecha);
  }
}
```

Figure 45: Constructor Date con otra fecha

```
Fri Feb 02 15:32:12 CET 1990
```

¡Coincide! El único problema parece ser que poner una fecha en milisegundos no es lo más cómodo, para ello tenemos otro par de clases, que aunque no son tan usadas y no nos detendremos en visitarlas en profundiad conviene conocer, estas son las clases Calendar y GregorianCalendar.

2. Métodos y propiedades Los métodos más interesantes de la clase date son getTime() y setTime(long time), que nos permiten obtener o poner los milisegundos que han pasado desde Epoch.

9.4.6 List y ArrayList

Las clases `List` y `ArrayList` las trataremos juntas porque normalmente se usan unidas, se tratan de, como su propio nombre indica, dos clases que tratan listas de elementos, ambas clases pertenecen al paquete `java.util`.

La razón por la que normalmente se usan unidas, (List para la declaración y ArrayList para la asignación) es porque ArrayList es una `implementación` de List (que es una **interfaz**) y, si bien podríamos declarar directamente un objeto de la clase ArrayList, hay muchísimos métodos que devuelven List.

1. Declaración e inicialización Los objetos de la clase ArrayList se instancian de manera habitual con `new` seguido de su constructor, si bien, al ser listas, podemos indicar de que tipo son, esto se hace con lo que se conoce como *operador diamante* por la forma que tiene,

   Por ejemplo, para inicializar un objeto de la clase ArrayList declarándolo como List que contenga una lista de Strings haremos lo siguiente:

   ```
   List<String> miLista = new ArrayList<String>();
   ```

   Figure 46: Declaración List-ArrayList

   Como hemos dicho antes, no es necesario usar la interfaz List, podemos declarar la variable como ArrayList directamente:

   ```
   ArrayList<String> miLista = new ArrayList<String>();
   ```

   Y, por último, a partir de Java 7, tampoco es necesario especificar dos veces el tipo de la lista como vemos en los dos ejemplos anteriores, sino que se puede omitir en la instanciación dando lugar al famoso *operador diamante*:

   ```
   List<String> miLista = new ArrayList<>();
   ```

   Esta es la manera más común de instanciar un ArrayList.

   La clase ArrayList tiene otros métodos constructores en los que no entraremos en detalle aquí, por ejemplo, se puede crear una lista a partir de otra.

2. Métodos más comunes

   - size(): nos devuelve el tamaño de la lista.
   - isEmpty(): nos devuelve `true` si la lista está vacía.
   - contains(Objeto): nos devuelve `true` si el objeto está en la lista.

- toArray(): nos devuelve un array con los valores de la lista
- get(int indice): nos devuelve el elemento de la posición designada.
- add(Objeto): inserta un objeto al final de la lista.
- add(int indice, Objeto): inserta un objeto en la posición designada de la lista.
- remove(int indice): elimina el objeto en la posición designada de la lista.
- remove(Objeto): elimina el primer objeto que coincida con el que se pasa como parámetro.
- clear(): elimina todos los objetos de la lista.
- sublist(int inicio, int final): devuelve una lista con los objetos comprendidos entre los indices especificados.

9.4.7 System

La clase `System` es una de las primeras clases que usamos cuando empezamos a programar en java (¿Te suena el System.out.println()?) y nos provee de una serie de funcionalidades que normalmente tienen que ver con funciones del sistema (Entrada/Salida, variables de entorno, fechas,...).

Todos los métodos que provee esta clase son **estáticos** y no hace falta crear un objeto de la misma para utilizarlos.

1. Métodos y propiedades más comunes Las propiedades más comunes de la clase system son los archiconocidos `in`, `out` y `err`, de tipo InputStream (entrada) el primero y de tipo PrintStream (salida) los otros dos. Cada una de estas propiedades (al ser objetos de otras clases) tiene su propio conjunto de propiedades y métodos, pero sin entrar en detalle, los dos más conocidos y utilizados son los de los objetos de PrintStream (out y err) print() y println(), utilizados para imprimir por pantalla (salida estándar) sin y con un salto de línea al final respectivamente.

   En cuanto a los métodos, los más utilizados son:

   - currentTimeMillis(): devuelve un `long` con el número de milisegundos que han pasado desde Epoch hasta el instante actual.
   - nanoTime(): lo mismo pero con nanosegundos.

9.4.8 Scanner

Otra de las clases de java con las que primero nos encontramos es la clase Scanner del paquete `java.util`, se trata de un escáner de texto que normalmente se usa para introducir texto en nuestros programas.

1. Constructor Su constructor más conocido es el que acepta un objeto de la clase `InputStream` como parámetro y que normalmente se usa de la siguiente manera para introducir texto desde la consola:

   ```
   // Utilizamos System.in del que hemos hablado anteriormente como parámetro para el
   // System.in también es conocido como la entrada estándar, es decir, entrada de te
   Scanner scan = new Scanner(System.in);
   ```

   Tiene muchos más constructores, pero el más utilizado es el anteriormente señalado.

2. Métodos y propiedades Los métodos mas conocidos son los métodos *next* que devuelven un valor en base al método que se use (y al texto introducido) algunos son nextInt(), nextBoolean(), nextLong(), nextLine(), etc.

10 Herencia

La herencia es una característica de los lenguajes de programación orientados a objetos que nos permite reutilizar y extender muchas funcionalidades, a grandes rasgos, la herencia nos permite coger una clase que ya esté creada y añadirle más funcionalidad o cambiar (sobreescribir) la que ya existe.

Por ejemplo, supongamos que tenemos la clase Persona:

```
public class Persona {

  String nombre;
  String sexo;
  int edad;
  String dni;

  public Persona (String nombre, String sexo, int edad) {
    this.nombre = nombre;
    this.sexo = sexo;
```

```
  this.edad = edad;
}

public void saludar() {
  System.out.println("Hey!");
}

public void cumple() {
  System.out.println("Feliz cumpleaños " + getNombre() + "!");
  edad++;
}

// Getters y Setters

public String getNombre(){
  return nombre;
}

public void setNombre(String nombre) {
  this.nombre = nombre;
}

public String getSexo() {
  return sexo;
}

public void setSexo(String sexo){
  this.sexo = sexo;
}

public int getEdad(){
  return edad;
}

public void setEdad(int edad){
  this.edad = edad;
}

public String getDni() {
  return dni;
```

```
  }

  public void setDni(String dni){
    this.dni = dni;
  }


}
```

Figure 47: Persona

Y ahora supongamos que esa persona, además tiene un trabajo, por lo tanto se convierte en un empleado:

```
//Como todo empleado es una persona compartirá todas sus funciones con esta, por lo tan
public class Empleado extends Persona {

  //Añadimos más propiedades específicas de la clase
  long codigoEmpleado;
  String departamento;

  // En el método constructor necesitaremos también los atributos del constructor de la
  public Empleado (String nombre, String sexo, int edad, long codigoEmpleado, String de
    //Con la palabra reservada 'super' llamamos a la clase padre, en este caso al const
    super(nombre, sexo, edad);

    //Y asignamos los valores de la clase Empleado a sus propiedades como siempre
    this.codigoEmpleado = codigoEmpleado;
    this.departamento = departamento;
  }

  //Como ahora somos empleados, tendremos que saludar de una manera más formal, para el
  // el método saludar() de la clase Persona.
  @Override
  public void saludar() {
    System.out.println("Buenos días");
  }

  // Getters y Setters
  // No es necesario implementar los de la clase padre (nombre, sexo, edad y dni) porqu
  // cualquier objeto de esta clase por ser métodos públicos y esta extender de Persona
```

```
  public long getCodigoEmpleado() {
    return codigoEmpleado;
  }

  public void setCodigoEmpleado(long codigoEmpleado) {
    this.codigoEmpleado = codigoEmpleado;
  }

  public String getDepartamento() {
    return departamento;
  }

  public void setDepartamento(String departamento) {
    this.departamento = departamento;
  }

}
```

Figure 48: Empleado

Entonces, si ahora creamos instancias de estas clases que acabamos de crear, podemos ver algo como lo siguiente:

```
public class Herencia {

  public static void main (String [] args) {

    Persona persona = new Persona("Manolito", "Hombre", 17);

    persona.saludar();
    System.out.println("La edad de " + persona.getNombre() + " es " + persona.getEdad()
    persona.cumple();
    System.out.println("La edad de " + persona.getNombre() + " es " + persona.getEdad()

    Empleado empleado = new Empleado("Manolita", "Mujer", 20, 1234567L, "Consultoría")

    // Podemos ver como el método saludar ha sido sobreescrito en Empleado
    empleado.saludar();

    // Comprobamos que los métodos que vienen de Persona y no han sido sobreescritos pu
```

```
    System.out.println("La edad de " + empleado.getNombre() + " es " + empleado.getEdad
    empleado.cumple();
    System.out.println("La edad de " + empleado.getNombre() + " es " + empleado.getEdad

  }

}
```

Figure 49: Herencia

```
Hey!
La edad de Manolito es 17
Feliz cumpleaños Manolito!
La edad de Manolito es 18
Buenos días
La edad de Manolita es 20
Feliz cumpleaños Manolita!
La edad de Manolita es 21
```

10.1 Interfaces

Antes de seguir hablando de herencia propiamente dicha hablaremos de las **interfaces**. Las interfaces son un tipo de documento en Java que no define ninguna funcionalidad, no hace nada, sino que se trata de un "contrato" que las clases implementadas pueden contraer o, en lenguaje Java **implementar**, esto se hace a través de la palabra reservada `implements` cuando declaramos la clase, por ejemplo, podemos declarar una interfaz que nos indique las funciones vitales de un animal:

```
public interface Animal {

  void comer();
  void respirar();
  void reproducirse();

}
```

Figure 50: Interfaz Animal

Ahora podemos crear varias clases que implementen esa interfaz, veremos que, como tiene 3 métodos cada una de las clases **necesita implementar**

esos 3 métodos, de ahí que antes considerásemos las interfaces como un contrato, ya que cualquier clase que las implemente necesita implementar también los métodos que ésta defina:

```
public class Gato implements Animal {

  @Overrides
  public void comer() {
    System.out.println("Comer croquetillas de mi cuenco");
  }

  @Overrides
  public void respirar() {
    System.out.println("Inhalar, exalar");
  }

  @Overrides
  public void reproducirse() {
    System.out.println("Irse por los tejados...");
  }

}
```

Figure 51: Clase Gato

Otro ejemplo sería el siguiente:

```
public class Tiburon implements Animal {

  public void comer() {
    System.out.println("Enseña esos dientes!");
  }

  public void respirar() {
    System.out.println("A ver esas branquias...");
  }

  public void reproducirse() {
    System.out.println("Esos huevos no se van a fecundar sólos...");
  }
```

```
}
```

Figure 52: Clase Tiburon

Como podemos ver ambas clases implementan los mísmos métodos de la misma interfaz pero ambas lo hacen de maneras diferentes (vale, en estos ejemplos son muy similares, pero te haces una idea).

Esto nos da una posibilidad muy interesante, como vimos en la explicación de los ArrayList estos se declaraban con la interfaz y luego se inicializaban con la clase que la implementa, pues ahora podemos hacer lo mismo:

```
public class Fauna {

  public static void main (String... args) {

    //Declaramos un objeto de tipo Animal
    Animal animal;

    // Recordemos que Animal es una interfaz y por lo tanto no se puede inicializar
    // (no tiene método constructor), así que necesitamos recurrir al de una clase que
    // implemente la interfaz
    animal = new Gato();

    // Ahora el objeto animal es una instancia de la clase Gato y tiene sus funciones
    animal.comer();
    animal.respirar();
    animal.reproducirse();

    // Pero el objeto sigue siendo de la "clase" Animal, así que le podemos dar otro va
    animal = new Tiburon();

    // Y ahora el objeto animal es una instancia de la clase Tiburon y por lo tanto, su
    animal.comer();
    animal.respirar();
    animal.reproducirse();

  }
```

Figure 53: Implementaciones de la interfaz Animal

10.2 Clases abstractas

10.3 Anotaciones

11 Tests

11.1 jUnit

12 Expresiones Lambda y Streams

12.1 Expresiones Lambda

12.2 Streams y programación funcional

https://www.javaworld.com/article/3314640/functional-programming-for-java-developers-p
html

13 Frameworks

13.1 Spring

13.1.1 Spring Boot

13.2 Hibernate y JPA