Universidad de La Laguna

Tema 3. Elementos de programación estructurada

CHRISTOPHER EXPÓSITO IZQUIERDO
AIRAM EXPÓSITO MÁRQUEZ
ISRAEL LÓPEZ PLATA
MARÍA BELÉN MELIÁN BATISTA
JOSÉ MARCOS MORENO VEGA

La programación estructurada

- La programación estructurada es aquel paradigma de programación que dice que todo programa puede escribirse únicamente utilizando 3 estructuras básicas de control:
  - Secuencial
  - Selección
  - Iteración

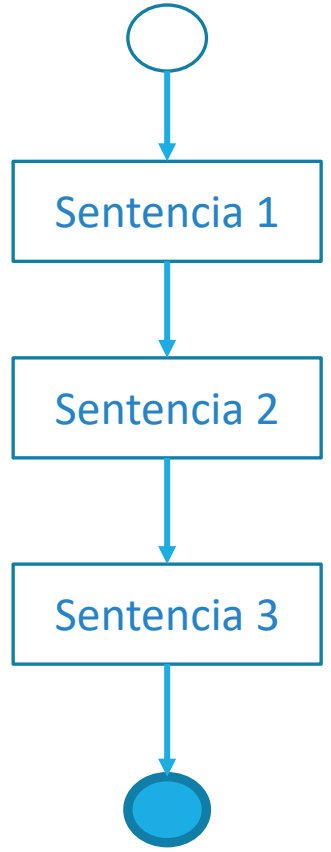
- Se basa en un teorema desarrollado por Boehm y Jacopini en 1966

- Define un problema como aquel compuesto por un determinado número de subproblemas mas sencillos que el original

- Forma una estructura **top-down**, es decir, jerárquica

La programación estructurada

- En programación estructurada, el diseño del programa es **descendente**.
- Las estructuras de control manejadas son limitadas a 3. Evita el uso de las sentencias **GOTO**
- El ámbito de las estructuras también se encuentra limitado
- Cualquier algoritmo en programación estructurada puede ser representado mediante un diagrama de flujo

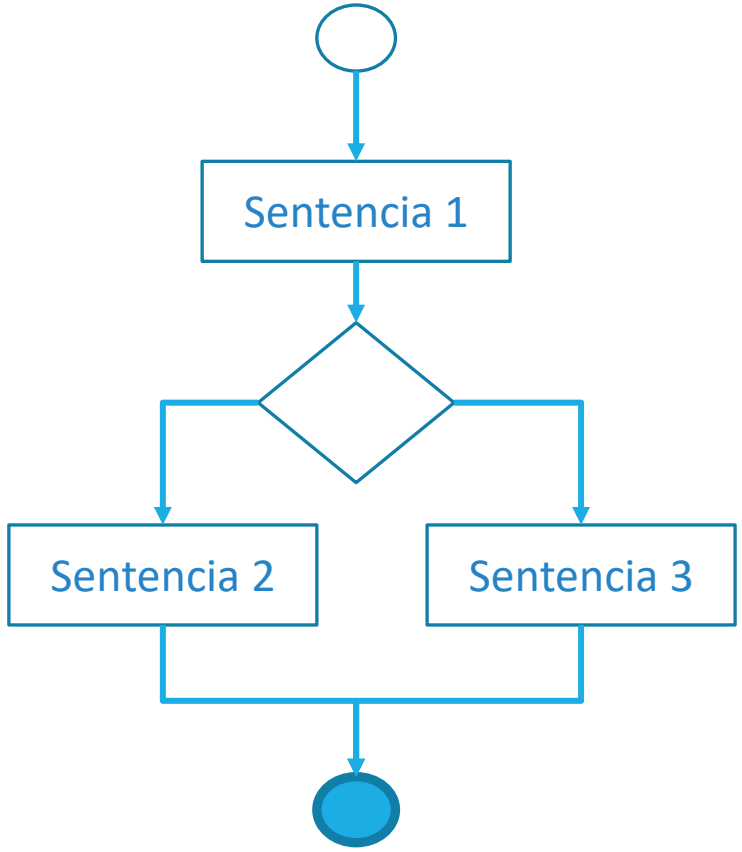


Ventajas

- El paradigma de programación estructurada está pensado para mejorar la calidad, claridad y tiempo de desarrollo
- Los programas pueden leerse de forma secuencial, por lo que son más fáciles de entender
- La estructura de los programas es muy clara
- Reduce el esfuerzo en pruebas. Facilita el seguimiento de los errores del programa gracias a su estructura mas sencilla
- El software creado es mas fácil de ampliar, ya que es mas sencillo
- Se elimina el **código espagueti**

Ejecución de un programa

- El flujo de ejecución de un programa se realiza de manera lineal
- Puede ser representado mediante un diagrama de flujo
- Existen estructuras que permiten cambiar esta linealidad
  - Condicionales.
  - Bucles.



Condicionales

- Permite ejecutar un conjunto de sentencias si se cumplen una serie de condiciones lógicas

- Una condición lógica se puede considerar como una comprobación mediante operadores relacionales que puede dar valores de **true** o **false**

- Fundamental a la hora de desviar el flujo del programa

Condicionales

- Las condiciones lógicas evalúan si unas determinadas variables o constantes cumplen con una determinada restricción

- Se pueden anidar las diferentes evaluaciones mediante operadores lógicos

Operador relacional		Operador lógico	
<	Menor que	&&	And
>	Mayor que	||	Or
<=	Menor o igual que	!	Not
>=	Mayor o igual que		
==	Igual que		
!=	Distinto a		

Condicionales

- And (&&)

Op1	Op2	Resultado
false	false	false
false	true	false
true	false	false
true	true	true

- Or (||)

Op1	Op2	Resultado
false	false	false
false	true	true
true	false	true
true	true	true

- Not (!)

Op1	Resultado
false	true
true	false

Condicionales

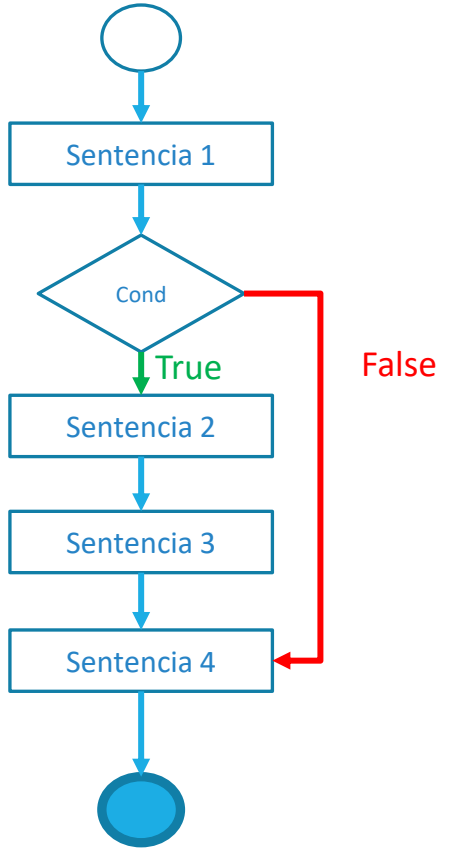
- Gracias a los operadores lógicos, se puede hacer un anidamiento de condiciones lógicas

- En este caso, es importante tener en cuenta la precedencia de los operadores

- **Ejemplo:** int a = 3; int b = 5; int c = 3; boolean d = true;
  - (a == c) && (a < b) && d 
  - (a == c) && (a < b) && !d 
  - (a == c) && (a < b) || !d 
  - (a == c) && (a > b) || !d 
  - ((a == c) && (a > b)) || !d 

Condicional if

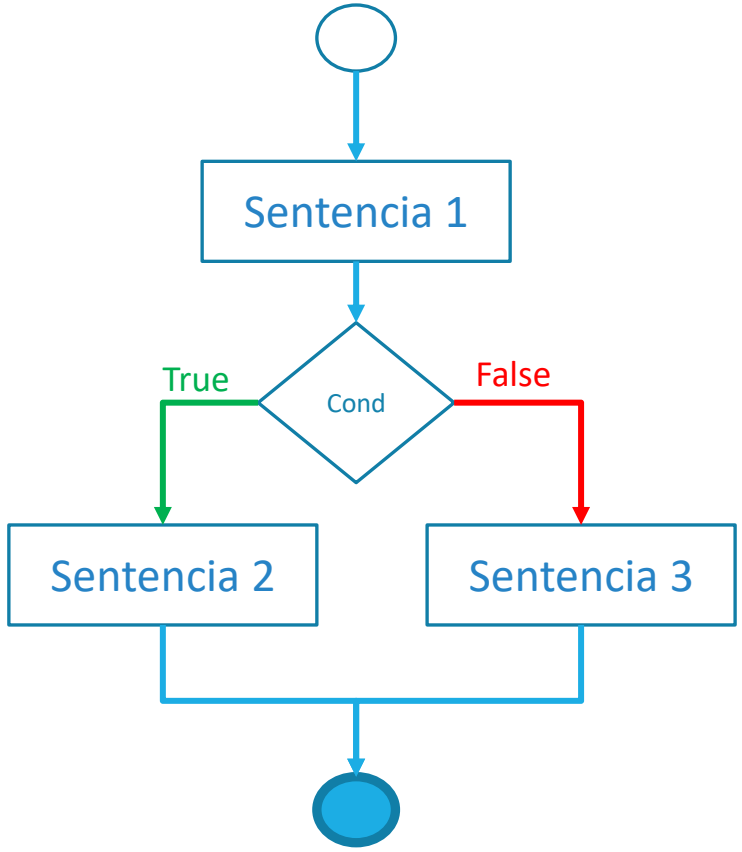
- **If:** Ejecuta un conjunto de sentencias sólo si se da una condición



Condicional if-else

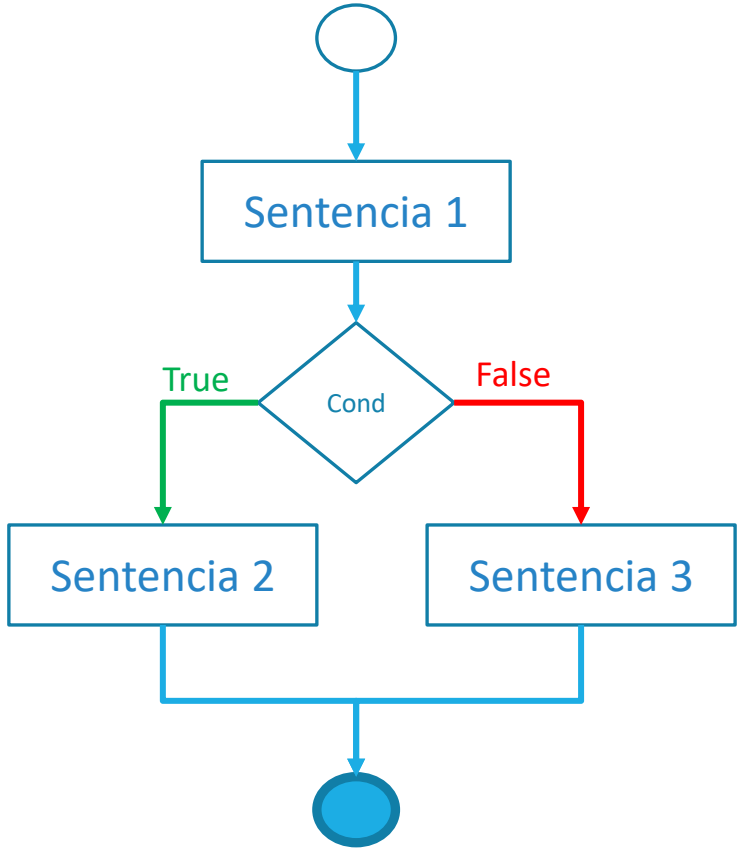
- **If:** Ejecuta un conjunto de sentencias sólo si se da una condición

- **If-Else:** Igual que el if, pero con unas sentencias alternativas. Se pueden concatenar varios if y else



Condicional if-else

- Un if se define por la sentencia **if(condición lógica) {}**
- El bloque delimitado por llaves será el conjunto de sentencias que se ejecuten si la condición lógica tiene valor **true**
- Opcionalmente, se puede añadir después del bloque del if la sentencia **else {}**, cuyo bloque es el conjunto de sentencias ejecutadas si el valor no se cumple
- Existe la opción de incluir la sentencia **else if(condición lógica) {}**, cuyo bloque se ejecuta si la sentencia if anterior no se cumple y la condición lógica representada si

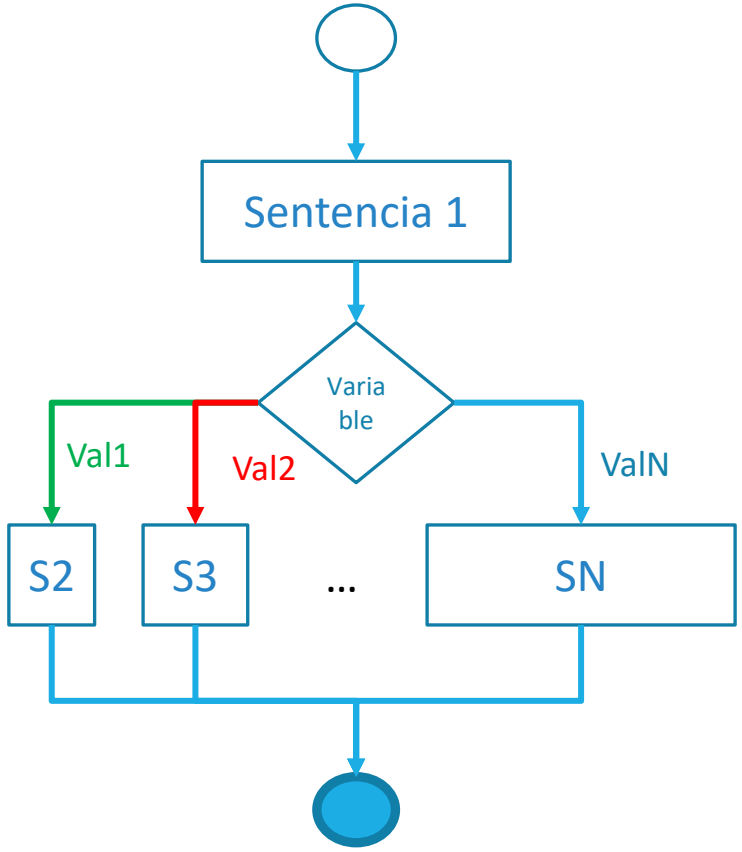


Condicional switch

- **If:** Ejecuta un conjunto de sentencias sólo si se da una condición

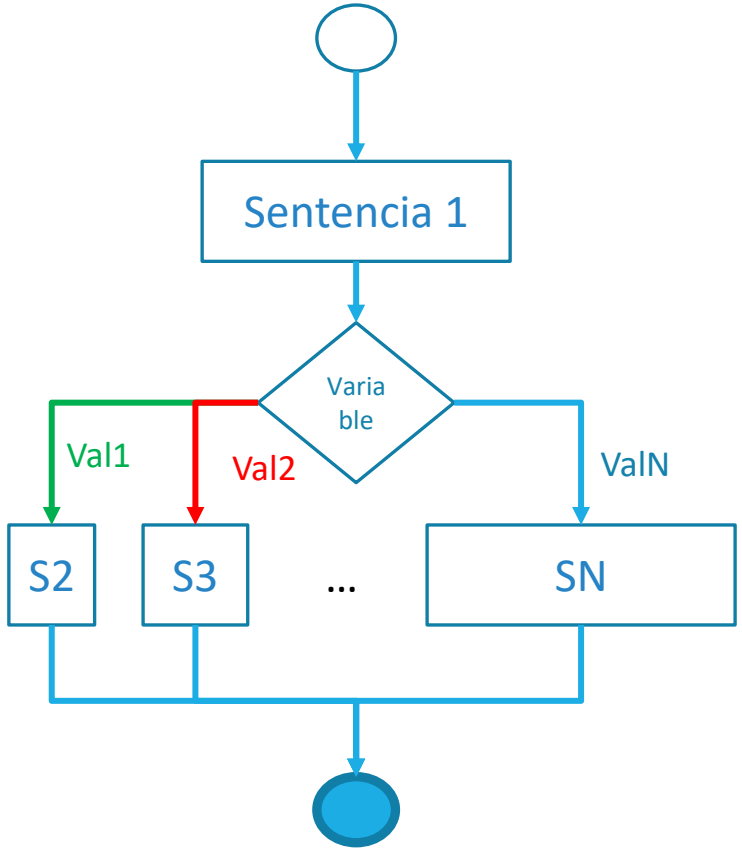
- **If-Else:** Igual que el if, pero con unas sentencias alternativas . Se pueden concatenar varios if y else

- **Switch:** Ejecuta un conjunto de sentencias dependiendo del valor de una variable



Condicional switch

- Las condiciones se ejecutan dependiendo del valor de una variable
- Se representa mediante la sentencia **switch(variable) {}**, siendo la variable entre paréntesis aquella cuyo valor se analiza
- Dentro del bloque del switch, se tienen las sentencias **case valor: {}**, cuyo bloque se ejecuta si la variable analizada tiene ese valor. Cada bloque del case debe terminar con la sentencia **break;**
- Siempre se debe incluir la sentencia **default: {}**, cuyo bloque se ejecuta cuando la variable no tiene ningun valor de los “cases” anteriores



Condicionales en Java

If

```
if (a < b) {
   System.out.println(“A es menor”);
   System.out.println(“B es mayor”);
}
```

Muestra los textos solo si el valor de la variable a es menor al de la variable b

If-Else

```
if (a < b) {
   System.out.println(“A es menor”);
} else {
   System.out.println(“A es mayor o igual”);
}
```

Muestra “A es menor” si el valor de la variable a es menor al de la variable b. En caso contrario, muestra “A es mayor o igual”

```
if (a < b) {
   System.out.println(“A es menor”);
} else if (a > b) {
   System.out.println(“A es mayor”);
} else {
   System.out.println(“SON IGUALES!”);
}
```

Ejemplo de concatenación de if y else

Switch

```
switch(a) {
   case 1: {
     System.out.println(“Opción 1”);
     break;
   }
   case 2: {
     System.out.println(“Opción 2”);
     break;
   }
   default: {
     System.out.println(“Resto”);
   }
}
```

Si la variable a vale 1, muestra “Opción 1”, si vale 2, muestra “Opción 2”. Para el resto de casos, muestra “Resto”.

Ejercicios de condicionales

1. Dado 2 números, mostrar el texto “Son iguales” sólo si los números son los mismos

2. Dados 2 números a y b, mostrar:
    1. “los 2 son negativos” si a y b son menores a 0
    2. “a es positivo, b es negativo” si a es mayor que 0 y b menor
    3. “a es negativo, b es positivo” si a es menor que 0 y b mayor
    4. “los 2 son positivos” si a y b son mayores a 0

3. Escribir el valor en letras de un número del 1 al 5 guardado en la variable **a.** En caso de no estar en este rango, escribir “Número incorrecto”

Sentencias repetitivas (bucles)

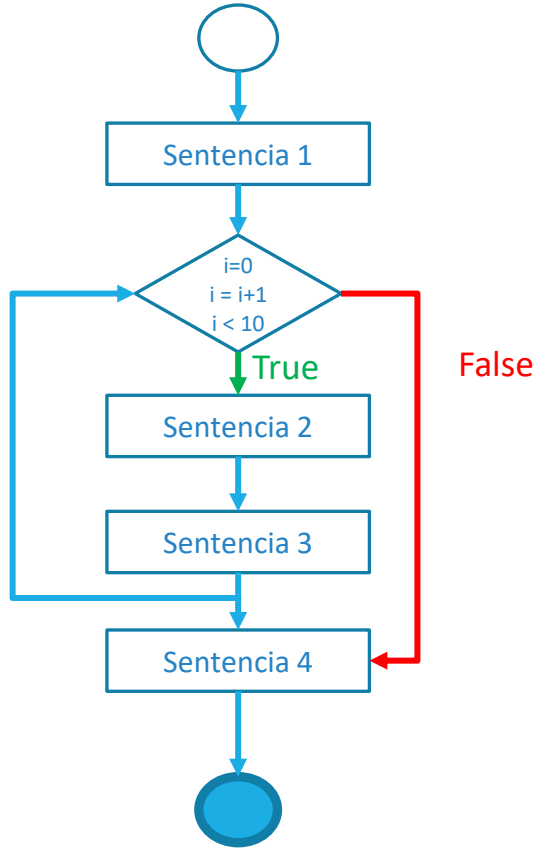
- Permiten repetir un bloque de sentencias un número determinado de veces
- El número de veces que se repiten las sentencias de un bucle dependen de una condición lógica
- Muy utilizados cuando se desea un comportamiento repetitivo del programa
- Ahorra la escritura de código repetido
- Ejemplo:
  - Escribe los números del 1 al 10
  - Escribe los números del 1 al 1000000

Bucle for

- Repite un conjunto de instrucciones un número predeterminado de veces

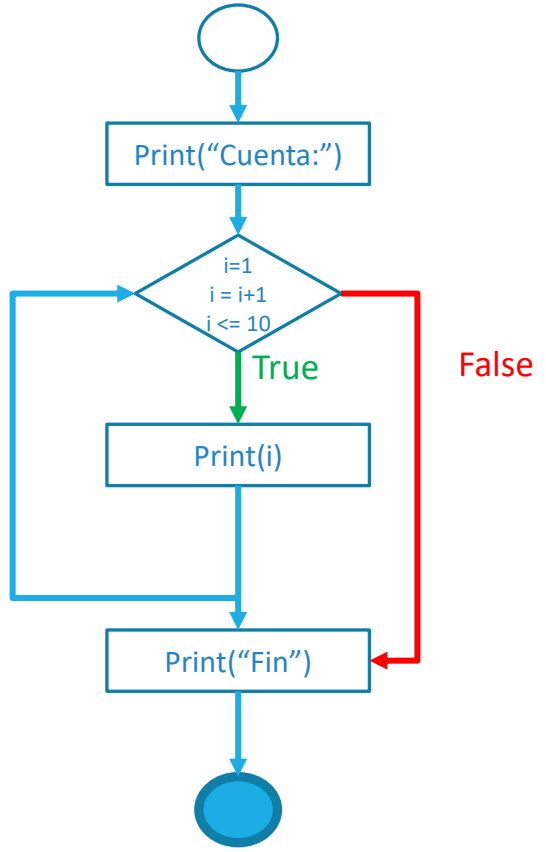
- El control de las iteraciones se realiza con una variable interna del bucle

- En este bucle se define:
  - Valor inicial de la variable
  - Incremento de la variable en cada iteración
  - Condición en la que termina el bucle



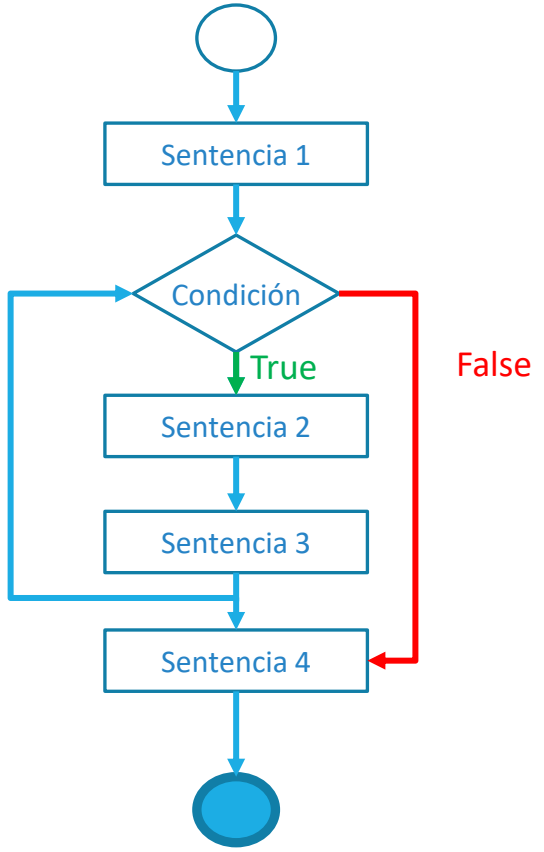
Ejemplo bucle for. Contar hasta 10

```
Cuenta:
1
2
3
4
5
6
7
8
9
10
Fin
```



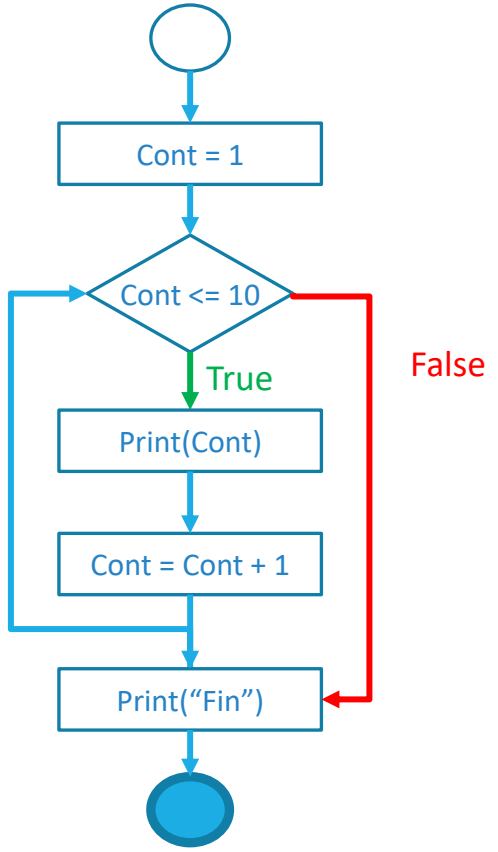
Bucle while

- Repite un conjunto de instrucciones siempre que se cumpla una condición lógica definida
  - Si la condición a evaluar tiene valor true, se ejecuta una iteración de las sentencias del bucle
  - Si la condición tiene valor false, se termina el bucle
- Si la condición no se cumple desde un inicio, las sentencias dentro del bucle nunca se ejecutan
- Si la condición se cumple siempre, se pueden ocasionar bucles infinitos



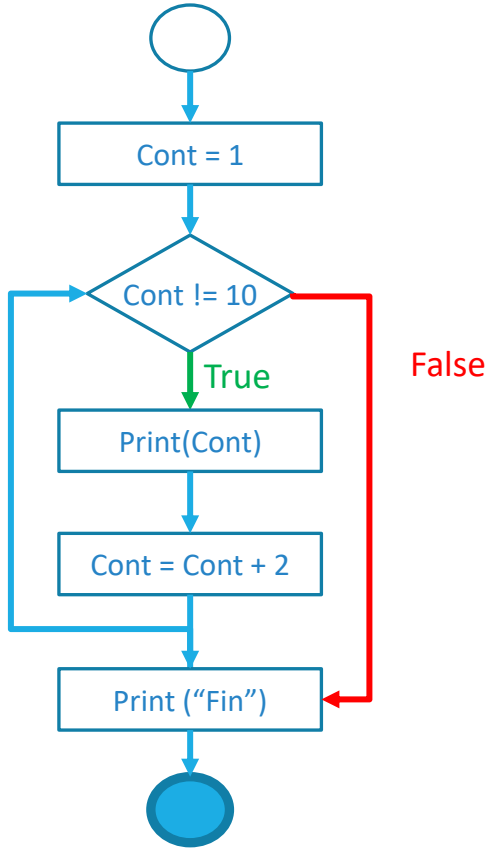
Ejemplo bucle while. Contar hasta 10

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



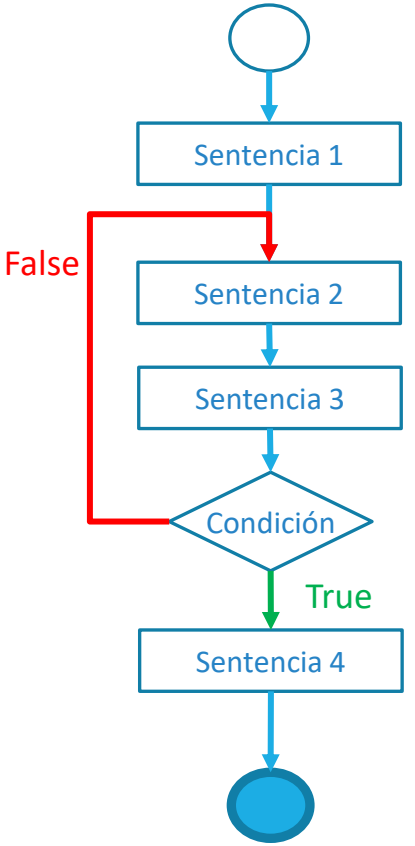
Ejemplo bucle while infinito

```
1
3
5
7
9
11
13
15
…
∞
```



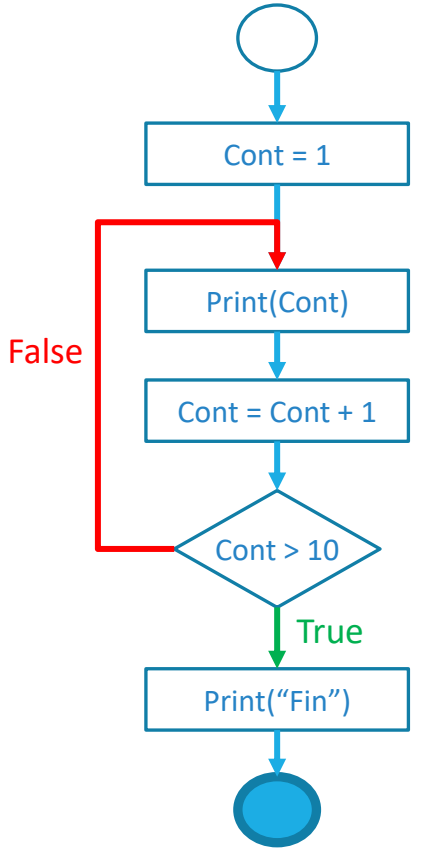
Bucle do-while

- Repite un conjunto de instrucciones siempre que se cumpla una condición definida
  - Si la condición a evaluar tiene valor false, se termina el bucle
  - Si la condición tiene valor true, se ejecuta una iteración de las sentencias del bucle
- Similar al bucle while. 1 diferencia:
  - La condición se comprueba al **final** de la estructura. En el bucle while es al inicio
- Al igual que el while, puede ocasionar bucles infinitos



Ejemplo bucle do-while. Contar hasta 10.

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



Bucles en Java. Ejemplo contar hasta 10

For

```
for (int i = 1; i <= 10; i++) {
   System.out.println(i);
}
System.out.println(“Fin”);
```

While

```
int i = 1;
while (i <= 10) {
   System.out.println(i);
   i++;
}
System.out.println(“Fin”);
```

Do-While

```
int i = 1;
do {
   System.out.println(i);
   i++;
} while (i <= 10);
System.out.println(“Fin”);
```

Bucles. Comparativa

Bucle for

- Utilizado cuando se quiere un número de iteraciones concreto
- Mejor opción para el recorrido de estructuras de datos

```
for (int i = 1; i <= 10; i++) {
   System.out.println(i);
}
System.out.println("Fin");
```

Bucle while

- Utilizado cuando la condición de parada es compleja
- Sólo ejecuta las sentencias mientras se cumpla la condición
- La condición se evalúa antes de las sentencias

```
int i = 1;
while (i <= 10) {
   System.out.println(i);
   i++;
}
System.out.println("Fin");
```

Bucle do-while

- Utilizado cuando la condición de parada es compleja y se quiere al menos una ejecución
- Sólo ejecuta las sentencias mientras se cumpla la condición
- La condición se evalúa después de las sentencias

```
int i = 1;
do {
   System.out.println(i);
   i++;
} while (i <= 10);
System.out.println("Fin");
```

break y continue

- Permiten romper un bucle de distintas formas
- **break:** Si se ejecuta la sentencia break, la ejecución del bucle termina

```
for (int i = 1; i <= 10; i++) {
   if (i == 5) break;
   System.out.println(i);
}
```

```
Muestra los números del 1 al 4
```

- **continue:** Si se ejecuta la sentencia continue, la iteración actual termina, pero el bucle continua

```
for (int i = 1; i <= 10; i++) {
   if (i == 5) continue;
   System.out.println(i);
}
```

```
Muestra los números del 1 al 10,
excepto el 5
```

Ejercicios de sentencias repetitivas

1. Escribir los números del 1 al 100 que sean impares

2. Escribir los números incrementando desde 1 hasta que la suma de todos esos números sea mayor a una variable entera x. Ejemplo:
   1. X = 10; 1, 2, 3, 4, 5 → 1+2+3+4+5 = 15
   2. X = 25; 1, 2, 3, 4, 5, 6, 7, 8 → 1+2+3+4+5+6+7+8 = 36

Buenas prácticas

- La indentación debe ser clara para los bucles. Se debe distinguir que sentencias están dentro de que bucle
  - Las llave abierta en un bucle se pone a la derecha de la condición o debajo. La llave cerrada, en su propia línea

- **OJO** a los bucles infinitos

- En un bucle for el número de la iteración lo tenemos en la variable interna. No usar otra para el mismo cometido

- En un condicional switch:
  - Incluir la opción default aunque no se utilice
  - Todas las opciones de un switch deben finalizar en un break

Estructuras de datos

- Una estructura de datos es un conjunto de valores (normalmente del mismo tipo) agrupadas bajo un mismo identificador

- Sirven para representar agrupaciones de forma lógica, así como para facilitar su acceso

- En Java existen un gran número de estructuras de datos para las distintas necesidades de almacenamiento de datos

- Dominar el manejo de las estructuras de datos es una tarea fundamental para resolver cualquier problema

Vectores unidimensionales

- Conjunto de datos agrupados e identificados por un mismo nombre
- Pueden ser de los mismos tipos de datos que cualquier variable (int, char, double, etc.)
- Se declaran siguiendo la forma **tipo[] nombre = new tipo[tamaño];**

```
int[] numeros = new int[10];
```

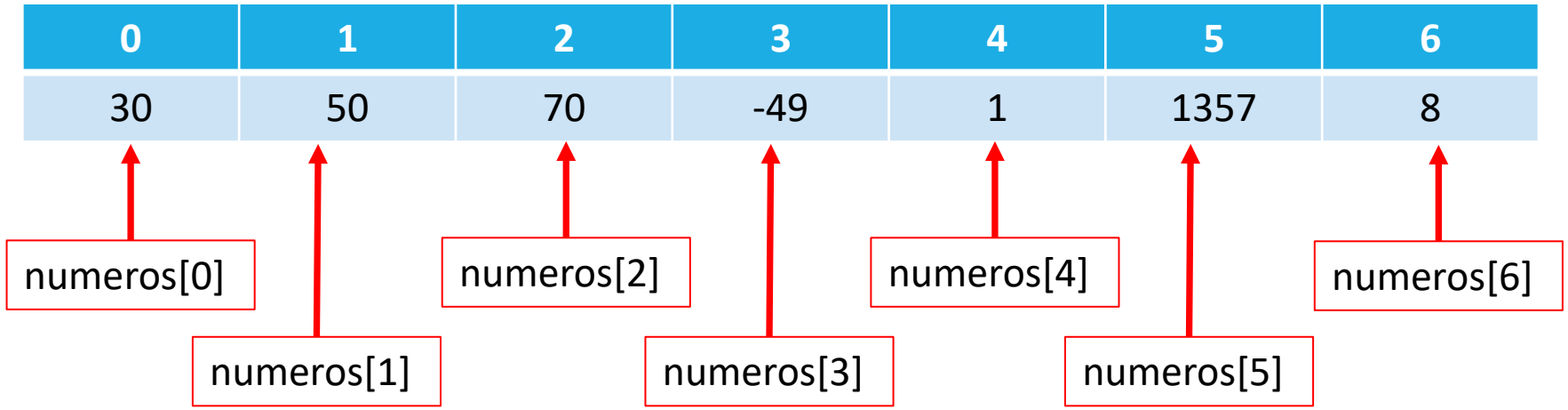
- El tamaño debe ser una constante entera
- A cada elemento se accede a través de un índice, empezando por el 0
- Pueden ser inicializados directamente. Ejemplo:

```
int[] numeros = {1, 2, 3};
```

- Los vectores simples se caracterizan por ser:
  - Unidimensionales
  - Tamaño fijo

Vectores unidimensionales. Ejemplo

```
int numeros[7] = {30, 50, 70, -49, 1, 1357, 8};
```



Vectores multidimensionales

- Extienden los vectores unidimensionales a 2 o mas dimensiones

- Se declaran siguiendo la forma **tipo[][] nombre = new tipo[tamaño1][tamaño2];**, donde cada grupo de corchetes representa una dimensión del vector

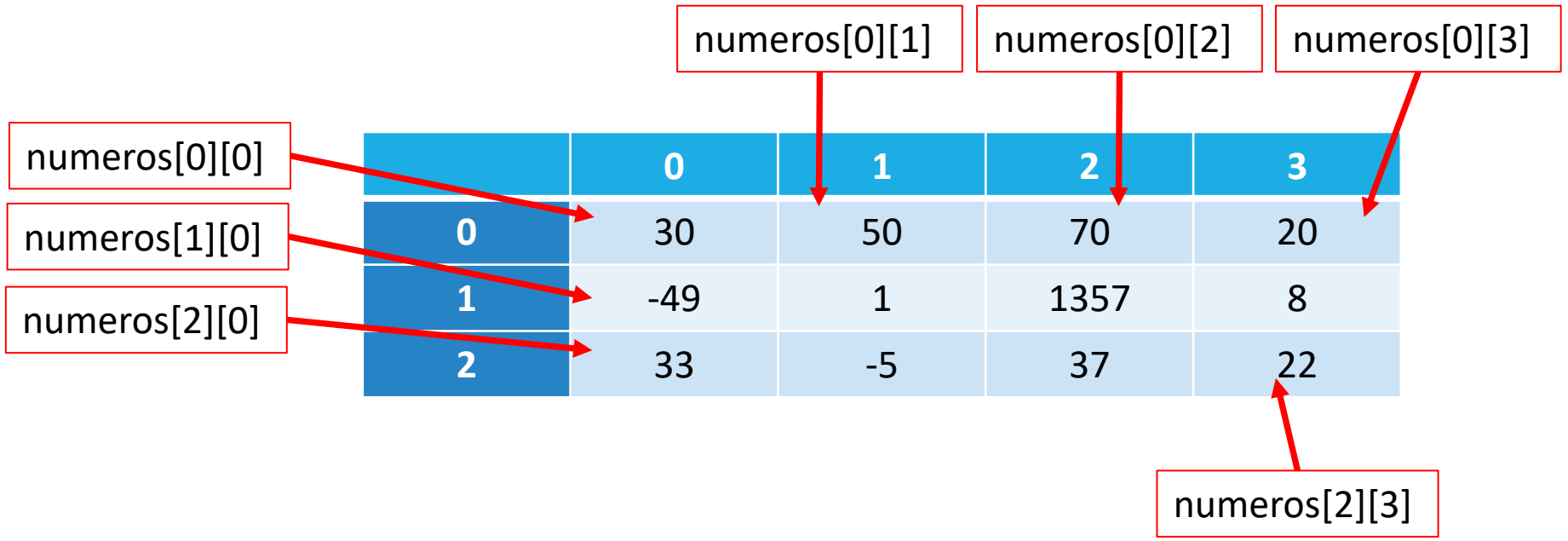
```
int[][] numeros = new int[10][15];
```

- Pueden ser inicializados directamente al igual que los vectores unidimensionales. Ejemplo:

```
int[][] numeros = {{1, 2, 3},
                   {4, 5, 6}};
```

Vectores multidimensionales. Ejemplo

```
int numeros[3][4] = {{30, 50, 70, 20} , {-49, 1, 1357, 8}, {33, -5, 37, 22}};
```



	0	1	2	3
0	30	50	70	20
1	-49	1	1357	8
2	33	-5	37	22

Ejemplo de bucles con vectores

```
/*Partimos de la existencia de un vector vec de enteros. Queremos mostrar sus valores de la forma
[indice] = valor */

//Tamaño del vector
int tam = vec.length;
//Con un bucle for
for (int i = 0; i < tam; i++) {
   System.out.println("[" + i + "] = " + vec[i]);
}
//Con un bucle while
int i = 0;
while (i < tam) {
   System.out.println("[" + i + "] = " + vec[i]);
   i++;
}
```

Ejercicios de estructuras de datos

1. Leer todos los elementos de un vector simple de enteros y de tamaño 10

2. Leer todos los elementos de un vector simple de enteros y de tamaño 10. Cuando uno de los elementos sea mayor a 20, dejar de leer.

3. Sumar todos los elementos de una matriz de tamaño 5x10

La clase String

- Como se ha visto, la clase String representa una cadena de caracteres

- Es una forma simplificada de representar un vector unidimensional de char.

- String cadena = “¡Hola!”; → char[] cadena = {‘¡’, ‘H’, ‘o’, ‘l’, ‘a’, ‘!’};

0	1	2	3	4	5
¡	H	o	l	a	!

- cadena[2] = ‘o’

La clase String

- La clase String nos permite realizar muchas operaciones con cadenas de caracteres como (partiendo del ejemplo String cadena = “Hola clase”;):
  - **Concatenación:** cadena + “ “ + “¿Qué tal?” → “Hola clase ¿Qué tal?”
  - **Conocer el tamaño:** cadena.size() → 10
  - **Obtener subcadenas:** cadena.substring(5, 9) → “clase”
  - **Dividir cadena:** String[] palabras = cadena.split(‘ ’) → palabras[0] = “Hola”, palabras[1] = “clase”
  - Y muchas mas cosas

- Java tiene una documentación muy completa de lo que se puede hacer con sus cadenas de caracteres en https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

La clase Object

- En Java existe una clase especial llamada Object, que representa a cualquier clase del sistema
- Implícitamente, todas las clases en Java derivan de Object
- Puede utilizarse para representar cualquier elemento genérico

```
Object obj;
obj = 1;
System.out.println(obj);
obj = “Prueba”;
System.out.println(obj);
```

Las clases contenedoras

- En Java, los tipos básicos (int, double, char, etc.) tienen asociados un conjunto de clases que extienden sus funcionalidades llamadas **clases contenedoras**

- Estas clases son muy parecidas en nombre a los tipos básicos asociados:

Primitivos	Clase Contenedora	Argumentos del Constructor
boolean	Boolean	boolean o String
byte	Byte	byte o String
char	Character	char
double	Double	double o String
float	Float	float, double o String
int	Integer	int o String
long	Long	long o String
short	Short	short o String

Las clases contenedoras

Funcionalidades

- ParseXXX(Object param): Convierte el elemento pasado por parámetro en el tipo del contenedor

```
Integer num = Integer.ParseInt(“32”);
```

```
num = 32
```

- XXXValue(): Convierte el elemento de la clase contenedora al tipo indicado en el nombre de la función

```
Integer num = new Integer(32);
String num = Integer.StringInt();
```

```
num = “32”
```

- ValueOf(): Convierte al tipo de la clase contenedora realizando las modificaciones necesarias

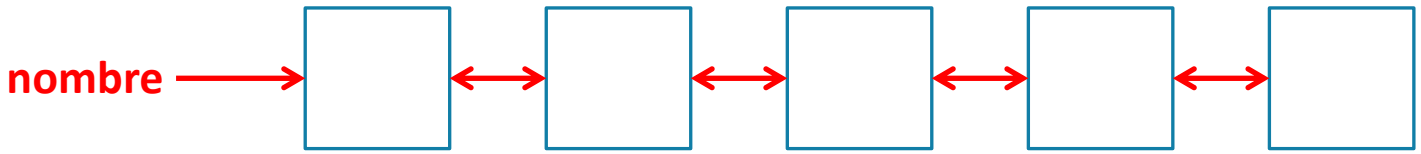
```
Integer num = Integer.valueOf(“101011”, 2);
```

```
num = 43
```

Otras estructuras de datos

- Java tiene un gran número de librerías con las que manejar distintas estructuras de datos
- Facilitan en gran medida la labor al programador
- **Vector dinámico:** Vector que no tiene un tamaño fijo, ya que varía según el número de elementos que se le han insertado

```
ArrayList<tipo> nombre = new ArrayList<>();
```

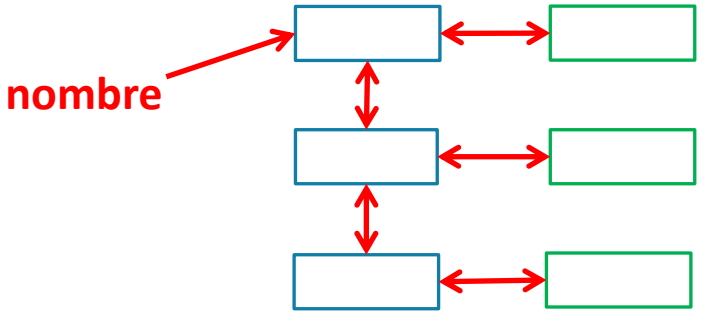


- **Enlace**: https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Otras estructuras de datos

- **Mapa:** Permite almacenar los datos en parejas de clave-valor
- Los valores se buscan en el mapa a partir de la clave

```
HashMap<tipoClave, tipoValor> nombre = new HashMap<>();
```

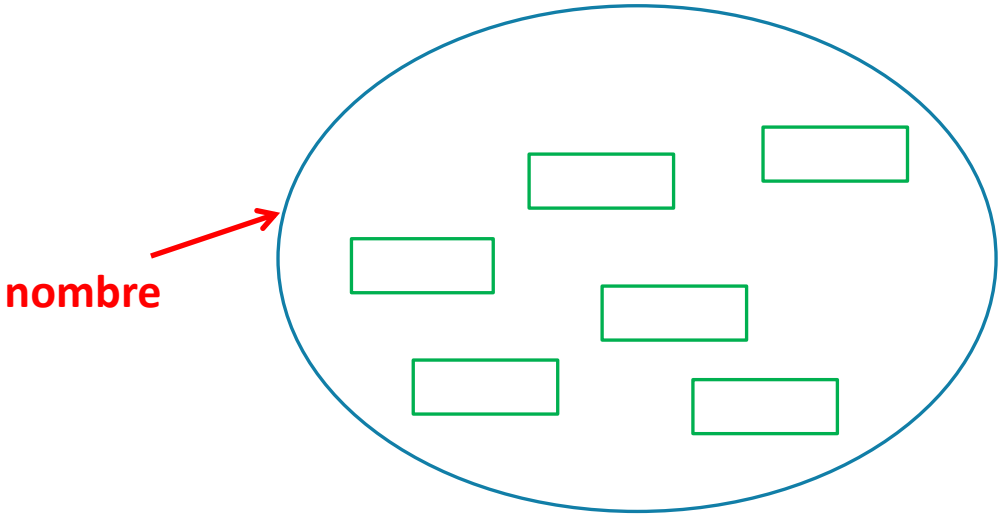


- **Enlace**: https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

Otras estructuras de datos

- **Conjuntos:** Almacena elementos, sin un orden determinado. No permite elementos repetidos

```
HashSet<tipo> nombre = new HashSet<>();
```



- **Enlace**: https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html