

Introducción a la Programación

Tema 2. Elementos del lenguaje Java

1. Elementos básicos del lenguaje	1
2. Tipos primitivos, clases envoltura y declaraciones	3
3. Tipos agregados: Array, List, String y Set.....	6
4. Expresiones y operadores	14
5. Sentencias de control	21
5.1. Sentencia if-else	21
5.2. La sentencia switch.....	25
5.3. Sentencia while	27
5.4. Sentencia for clásico	28
5.5. Sentencia for-each o for extendido	30
5.6. Sentencias continue y break.....	32
6. Relaciones entre interfaces y clases: herencia, implementación y uso. Grafo de tipos ..	33
7. Gestión de excepciones	37
8. Semántica del paso de parámetros en Java	43
9. Tipos genéricos y métodos genéricos	46
10. Conceptos Aprendidos	49
11. Ejercicios	50

1. Elementos básicos del lenguaje

Alfabeto del lenguaje

Conjunto de símbolos del estándar Unicode que establece hasta 65.535 signos de todas las lenguas principales, siendo los códigos del 0 al 255 los correspondientes a la codificación ASCII.

Lexemas del lenguaje

Unidad (indivisible o mínima) con significado. Sucesión finita de símbolos del alfabeto (uno o más)

Tipos de lexemas

Identificadores, palabras reservadas, literales, separadores y símbolos de operación.

Identificadores

Son palabras que permiten referenciar los diversos elementos que constituyen el código. Se construyen mediante una secuencia de letras, dígitos, o los símbolos `_` y `$`. En cualquier caso se debe observar:

- No pueden coincidir con **palabras reservadas** de Java (ver más adelante)
- Deben comenzar por una letra, `_` o `$`.
- Pueden tener cualquier longitud.
- Son sensibles a las mayúsculas, por ejemplo, el identificador `min` es distinto de `MIN` o de `Min`.

Ejemplos de identificadores válidos son los siguientes:

```
tiempo, distancial, caso_A, PI, velocidad_de_la_luz
```

Por el contrario, los siguientes nombres no son válidos (¿Por qué?)

```
1_valor, tiempo-total, dolares$, %final
```

En general, es muy aconsejable elegir los nombres de los identificadores de forma que permitan conocer a simple vista qué representan, utilizando para ello tantos caracteres como sean necesarios. Esto simplifica enormemente la tarea de programación y –sobre todo– de corrección y mantenimiento de los programas. Es cierto que los nombres largos son más laboriosos de teclear, pero, en general, resulta rentable tomarse esa pequeña molestia. Unas reglas aconsejables para los identificadores son las siguientes:

- Las variables normalmente tendrán nombres de sustantivos y se escribirán con minúsculas, salvo cuando estén formadas por dos o más palabras, en cuyo caso, el primer carácter de cada palabra se escribirá en mayúscula. Por ejemplo: `salario`, `salarioBase`, `edadJubilacion`.
- Los identificadores de constantes (datos que no van a cambiar durante la ejecución del programa) se deben escribir con todos los caracteres con mayúsculas. Por ejemplo: `PI`, `PRIMER_VALOR`, `EDADADMINIMA`.
- Los identificadores de métodos tendrán la primera letra en minúscula, y la primera letra de las siguientes palabras en mayúscula: `getX`, `setX`, `incrementarVolumen`, etc.
- Los identificadores de clases e interfaces se deben escribir con el primer carácter de cada palabra en mayúsculas y el resto en minúsculas: `Punto`, `PuntoImpl`, etc.

Palabras reservadas de Java

Una palabra reservada es una palabra que tiene un significado especial para el compilador de un lenguaje, y, por lo tanto, no puede ser utilizada como identificador. El conjunto de palabras reservadas que maneja Java se muestra en la siguiente tabla:

Palabras reservadas del lenguaje Java					
<code>abstract</code>	<code>Boolean</code>	<code>break</code>	<code>byte</code>	<code>case</code>	<code>catch</code>
<code>char</code>	<code>class</code>	<code>const</code>	<code>continue</code>	<code>default</code>	<code>do</code>

double	each	else	extends	final	finally
float	for	goto	if	implements	import
int	instanceof	interface	long	native	new
package	private	protected	public	return	short
super	static	switch	synchronized	this	throw
throws	transient	try	void	volatile	while

Las [palabras clave](#) pueden consultarse en la documentación de Java.

Literales

Son elementos del lenguaje que permiten referenciar los distintos valores que pueden tomar los tipos del lenguaje.

Ejemplo	Tipo
2, 0x1a, 0b11010	int
3.1415926	double
'a', '\t', '\n', '\r'	char
"rojo"	String
1000L	long
300.5f	float
false	boolean
null	Objeto

Comentarios

Los comentarios son un tipo especial de separadores que sirven para explicar o aclarar algunas sentencias del código, por parte del programador, y ayudar a su prueba y mantenimiento. De esta forma, se intenta que el código pueda ser entendido por una persona diferente o por el propio programador algún tiempo después. Los comentarios son ignorados por el compilador.

En Java existen comentarios de línea, que se marcan con `//`, y bloques de comentarios, que comienzan con `/*` y terminan con `*/`.

Ejemplo:

```
// Este es un comentario de una línea
/* Este es un bloque de comentario
   que ocupa varias líneas
*/
```

2. Tipos primitivos, clases envoltura y declaraciones

Tipos de datos Java

Como hemos comentado en el tema anterior, todos los lenguajes tienen unos tipos básicos.

Los **tipos de datos básicos, nativos o primitivos** de Java son:

- boolean - Pueden tomar los valores true o false
- byte
- int
- short
- char
- long
- double
- float
- void – Tipo que no tiene ningún valor

Tipos envoltura (wrappers)

Los tipos básicos son herencia de lenguajes de programación anteriores a Java. Por cada tipo básico se tiene disponible un tipo envoltura (*wrapper*):

- Byte para byte
- Short para short
- Integer para int
- Long para long
- Boolean para boolean
- Float para float
- Double para double
- Character para char

Clases envoltura y tipos primitivos – conversiones

Los tipos envoltura añaden funcionalidad a los tipos primitivos. Esta funcionalidad añadida y los detalles que diferencian a uno y otros los iremos viendo más adelante.

En general, un tipo primitivo y su correspondiente envoltura son completamente intercambiables (la conversión entre uno y otro es automática) y por ello usaremos preferiblemente los tipos envoltura en lugar de los tipos primitivos. A la conversión automática de un tipo primitivo a su envoltura se le denomina en inglés *autoboxing* y la inversa *unboxing*. Una diferencia importante entre ambos es que los tipos envoltura son objetos y los tipos primitivos no.

Los tipos envoltura, como todos los objetos, tienen constructores para crear objetos. Estos constructores tienen el mismo nombre que el tipo y diferentes posibilidades para los parámetros: un valor del tipo primitivo correspondiente, una cadena de caracteres que represente el valor, etc. Mediante un operador new seguido de uno de los constructores podemos crear objetos de tipo envoltura.

Ejemplo (declaración de variables de tipos envoltura y tipos básicos):

```
Integer a = 35;
Integer b = new Integer(10);
Integer c = new Integer("32");
int d = 14;
int e;
e=a+b+c+d;
```

En el ejemplo se muestran diferentes posibilidades de inicializar un tipo envoltura: *a* se inicializa a partir de un valor del tipo básico, *b* a partir de un constructor que toma un valor del tipo básico, *c* a partir de un constructor que toma una cadena de caracteres que representa un entero, *d* es un tipo primitivo y se inicializa de la única forma posible: mediante un valor.

El resultado de la suma de *a*, *b*, *c*, *d* es asignado a *e*. De forma similar se comportan los otros tipos envoltura.

Envolturas y concepto de inmutabilidad

Los objetos de tipos envoltura son inmutables. Un objeto inmutable se caracteriza por:

- Las propiedades son fijadas por el constructor cuando el objeto se crea. Estas propiedades, como su valor, no pueden variar. Los métodos set no existen o son innecesarios.
- Si existen métodos que realicen operaciones sobre las propiedades del objeto, el resultado es otra instancia del tipo que contiene los datos modificados. Esto tiene consecuencias importantes sobre todo cuando los tipos envolturas se pasan como parámetros como veremos más adelante.

Además de las envolturas existen otros tipos que también son inmutables, como por ejemplo las cadenas de caracteres o String.

Variables

Son elementos del lenguaje que permiten guardar y acceder a los datos que se manejan. Es necesario declararlas antes de usarlas en cualquier parte del código y por convenio se escriben en minúsculas. Mediante la declaración indicamos que la variable guardará un valor del tipo declarado. Mediante una asignación podemos dar un nuevo valor a la variable.

La sintaxis responde al siguiente patrón:

```
tipo nombredeVariable [=valor] [, nombredevariable...];
```

Ejemplo (declaración e inicialización de variables):

```
int valor;
Double a1= 2.25, a2= 7.0;
char c= 'T';
String cadena= "Curso Java";
```

NOTA: String es un tipo de dato NO nativo de Java que se estudiará más adelante.

Constantes

Son elementos del lenguaje que permiten guardar y referenciar datos que van a permanecer invariables durante la ejecución del código. La declaración de una constante comienza por la palabra reservada *final*.

Es necesario declararlas y por convenio se hace con todas sus letras en mayúsculas.

La sintaxis responde al siguiente patrón:

```
final tipo_de_dato NOMBREDECONSTANTE = valor;
```

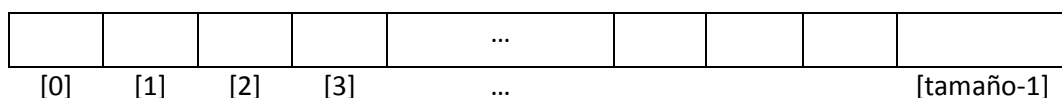
Ejemplo (declaración de constantes):

```
final int DIAS_SEMANA = 7;
final Double PI = 3.1415926;
final String TITULO = "E.T.S. de Ingeniería Informática";
```

En este ejemplo se declaran tres constantes, *DIAS_SEMANA*, *PI*, y *TITULO*, de tipo *int*, *Double* y *String*, respectivamente. Nótese que, por convención, los nombres de constantes se escriben en mayúsculas.

3. Tipos agregados: Array, List, String y Set

Java proporciona los tipos **array**, **List**, **String** y **Set** para gestionar agregados lineales de elementos del mismo tipo. Los objetos de los tipos *array*, *List* y *String* son agregados lineales de elementos que pueden ser accedidos (indexados) a través de la posición que ocupan en la colección. El tipo *Set* no es indexable. Cada uno de estos agregados lineales tiene un **tamaño** que es el número de elementos que contiene. Los tipos indexables podemos considerarlos formados por un conjunto de celdas. Cada celda está identificada de forma única por un índice. Un índice es una variable de tipo *int* que representa la celda ubicada en una posición dada del agregado indexable. Los valores del índice van de 0 al tamaño menos 1. La primera posición es 0 y la última, es el tamaño menos 1. Cada celda del agregado indexable tiene un índice y un contenido. El contenido de la celda es un elemento del agregado. Un agregado indexable podemos representarlo por:



Como cualquier variable cada uno de los agregados tiene un tipo y con ese tipo podemos declarar variables. La **declaración** de un *array* de elementos de un tipo dado se hace

añadiendo [] al tipo. Los objetos de tipo *String* de la forma usual. Los objetos de tipo *List<T>* y *Set<T>* los declararemos poniendo un tipo concreto dentro de <>.

```
int[] a;
Integer[] b;
String s;
List<Float> v;
Set<Integer> c;
```

En el anterior ejemplo decimos que *a* es un *array* de *int*, *b* es un *array* de *Integer*, *s* es un *String*, *v* una *List* de *Float* y *c* un *Set* de *Integer*.

Todos los agregados tienen un conjunto de propiedades consultables que en algunos casos son también modificables. Veamos cada una de esas propiedades y el método concreto que las representa en cada agregado:

- **Tamaño:** Es el número de elementos que contiene el agregado
 - *Array*: El tamaño es inmutable y por lo tanto no cambia una vez creado el objeto. Se representa por el atributo público *length*.
 - *String*: El tamaño es inmutable y por lo tanto no cambia una vez creado el objeto. Se representa por el método *length()*.
 - *List<T>*: El tamaño es mutable pero no puede ser modificado directamente. Es decir el tamaño puede cambiar mediante algunas operaciones pero no puede ser modificado directamente. Se representa por el método *size()*
 - *Set<T>*: El tamaño es mutable pero no puede ser modificado directamente. Es decir el tamaño puede cambiar mediante algunas operaciones pero no puede ser modificado directamente. Se representa por el método *size()*
- **Contenido de una celda:** Solo disponible para los agregados indexables
 - *Array*: El contenido de una celda es consultable y modificable. La consulta y modificación se consiguen combinando la variable de tipo array con el índice mediante el operador indexación: el índice encerrado entre corchetes. Según que la expresión formada esté a la derecha o a la izquierda de una asignación será una operación de consulta o modificación del contenido de la celda.
 - *String*: El contenido de una celda es inmutable. La consulta del contenido de la celda puede hacerse mediante el método *charAt(int i)*.
 - *List<T>*: El contenido de una celda puede consultarse y modificarse. La consulta se hace con el método *get(int i)* que devuelve el objeto de tipo *T* contenido en la celda *i*. La modificación con el método *set(int i, T e)* que cambia el contenido de la celda *i* y pone en ella el objeto *e*.

Los elementos que puede contener un de agregado: *array*, *List*, *String* o *Set* tienen las siguientes limitaciones:

- Un *array* pueden contener elementos de tipos objeto y de tipos primitivos.
- Los tipos *Set* o *List* pueden contener sólo elementos de tipo objeto.
- Los elementos de un objeto de tipo *String* sólo pueden ser caracteres.

En resumen:

	Pueden contener tipos primitivos	Pueden contener tipos objeto	Se puede modificar el tamaño	Puede modificarse el contenido de una celda
array	Sí	Sí	No	Sí
List	No	Sí	Sí	Sí
Set	No	Si	Si	No
String	Sólo char	No	No	No

Hay por lo tanto diferentes tipos de operaciones que podemos hacer con cada una de los agregados anteriores: declaración, consulta de la longitud, acceso al contenido de una celda y modificación del contenido de una celda (en algunos casos). Como todas las variables las que agregados de elementos tras ser declaradas deben ser inicializadas. Veamos como hacerlo en cada caso.

La inicialización para el caso de un *array* puede hacerse de dos maneras: indicando entre llaves y separados por comas los valores iniciales del *array* o indicando solamente la longitud del *array* en la forma *new tipo[longitud]* pero no el contenido de las celdas. En el caso del tipo *String* la inicialización se hace dando un valor adecuado. En el caso de *List<T>* y *Set<T>* con el operador *new* y el constructor adecuado. Más adelante veremos otros constructores posibles.

Ejemplos:

```
int[] a = {2,34,5};
Integer[] b = new Integer[7];
String s = "Hola";
List<Float> v = new ArrayList();
Set<Integer> c = new HashSet();
```

En las inicializaciones anteriores *a* es un *array* de tres valores *int*, *b* un *array* de 7 valores de tipo *Integer* sin contenido inicial, *s* tiene el valor "Hola", *v* es una *lista* vacía de *Float* (su tamaño es cero) y *c* es un conjunto de enteros vacío. En el caso de la variable *b* debemos de acabar de inicializarla. Tal como está tiene definido (y por lo tanto consultable) su tamaño pero no los contenidos de las celdas. En el caso de la variable *a* está inicializado el tamaño y el contenido de las celdas. Para inicializar el contenido de las celdas de la variable *b* podemos hacerlo dándole valores a cada una de ellas.

```
b[0] = 14;
b[1] = -10;
```



```
b[2] = 1;
b[3] = 0;
b[4] = -2;
b[5] = 34;
b[6] = 24;
```

Como vemos los índices disponibles van de 0 a 6 puesto que el tamaño es 7.

Una vez que están inicializados podemos consultar su tamaño. Como hemos visto en el caso de un *array* se hace con el atributo público *length*, en el caso de un *String* con el método *length()* y en el caso de un *List* y *Set* con el método *size()*. La operación de consulta del tamaño es una operación sin efectos laterales y devuelve un valor de tipo *int*.

```
int r = a.length + s.length() + v.size()+c.size();
```

En el ejemplo anterior *r* toma como valor la suma del número de elementos de *a* (3), *s* (4), *v* (0) y *c*(0), es decir, 7. Obsérvese que el atributo *length* cuando es aplicado a un *array* no lleva paréntesis porque es un atributo público. En el caso de un *String* es un método, igual que el método *size()* para el caso de un *Vector*.

Como hemos visto arribas el **acceso al contenido de una celda** (en el caso de los agregados indexables) permite obtener el contenido de una celda posición indicando el número de la misma. Sólo está permitido acceder a las celdas que van de 0 a la longitud menos 1, si intentamos acceder a otra celda se producirá un error en tiempo de ejecución que no es detectado en tiempo de compilación. Acceder a la posición *i* de un array *a* se consigue con *a[i]*, para un *String* con el método *charAt(i)* y para un *Vector* con el método *get(i)*. La operación de acceso al contenido de una celda es una operación sin efectos laterales y su tipo es el tipo de los elementos contenidos en el agregado correspondiente.

La **modificación del contenido de una celda** no es posible para objetos de tipo *String* (ni para agregados no indexables como *Set*) pero sí para objetos de tipo *array* y *List*. Cambiar el contenido de la celda *i* en un array *a* por un nuevo valor *e* se consigue asignando *e* a *a[i]*. Es decir *a[i]=e*. En caso de la lista *v* usando el método *set(i,e)*.

```
a[1] = a[0];
```

En el ejemplo anterior *a[0]* es una operación de consulta porque está a la derecha del operador de asignación. Devuelve el valor 2. La operación *a[1]* es una operación de modificación de la celda 1 porque está a la izquierda del operador de asignación. A la celda 1 se le asigna el valor 2. El vector quedaría de la forma {2,2,5}.

En el caso de los agregado cuyo tamaño puede modificarse hay disponible operaciones para añadir y eliminar elementos. Como se ha señalado esto no es posible en los objetos de tipo *array* ni en los de tipo *String*. En los de tipo *List* y *Set* sí es posible usando los métodos *add(e)*, para añadir el elemento y *remove(e)*. En el caso de una lista el elemento añadido se colca al

final de la misma y se elimina, si hubiera varios iguales, el primero que encontremos recorriendo la lista según índices crecientes de sus celdas.

```
List<Double> v1 = new ArrayList();
v1.add(3.0);
v1.add(25.0);
v1.add(7.0);
v1.add(9.0);
v1.remove(1.0);
v1.remove(7.0);
Set<Double> s1 = new HashSet();
s1.add(3.0);
s1.add(25.0);
s1.add(7.0);
s1.add(9.0);
s1.remove(1.0);
s1.remove(7.0);
```

Después de las sucesivas operaciones el contenido de `v1` es `[3.,25.,9.]` y el de `s1` `{3.0,9.0,25.0}`. Podemos observar que el método `remove(e)` elimina el objeto `e` si está y si no está no hace nada. También podemos ver que la lista ha mantenido los objetos en el orden en que los fuimos añadiendo pero no existe un orden definido para los elementos dentro del conjunto.

	Declaración	Inicialización	Tamaño	Acceso a una celda	Modificación de una celda	Añadir y eliminar elementos
array	<code>int [] a =</code>	<code>a = {2,34,5};</code>	<code>a.length</code>	<code>a[i]</code>	<code>a[i] = e;</code>	No es posible
List<T>	<code>List<Float> v</code>	<code>v= new ArrayList();</code>	<code>v.size()</code>	<code>v.get(i)</code>	<code>v.set(i, e);</code>	<code>v.add(i, e);</code> <code>v.remove(e);</code>
Set<T>	<code>Set<Float> c</code>	<code>c = new HashSet();</code>	<code>c.size()</code>	No es posible	No es posible	<code>c.add(i, e);</code> <code>c.remove(e);</code>
String	<code>String s</code>	<code>s = "Hola";</code>	<code>s.length()</code>	<code>s.charAt(i)</code>	No es posible	No es posible

Otras funcionalidades del tipo String

Junto a la funcionalidad ya vista, el tipo *String* ofrece métodos para concatenar a la cadena dada otra produciendo una nueva cadena (*concat*), decidir si la cadena contiene una secuencia dada de caracteres (*contains*), buscar la primera posición de un carácter dado su valor entero (*indexOf*), obtener la subcadena dada por dos posiciones incluyendo `i1` y sin incluir `i2` (*substring*), sustituir el carácter `c1` en todas sus apariciones por el carácter `c2`, etc.

```
public final class String ... {
    public int length();
    public char charAt(int i);
    public String concat(String s);
    public boolean contains(CharSequence s);
    public int indexOf(int c);
```

```

    public String substring(int i1, int i2);
    public String replace(char c1, char c2);
    ...
}

```

Otras funcionalidades del tipo List y Set

Junto a la funcionalidad vista los tipos `Set<T>` y `List<T>` ofrecen métodos para decidir si el agregado está vacío (`isEmpty()`), para saber si contiene un objeto dado (`contains(e)`). En el tipo `List<T>` específicamente algunos como (`indexOf`) que devuelve el índice de la primera celda que contiene el elemento o `add(i,e)` que añade el elemento *e* en la casilla de índice *i* desplazando a la derecha las de índice superior. O igual.

Los interfaces `Set<T>` y `List<T>` están en el paquete `java.util` (es necesario importarlo para su uso). Incluimos las firmas de algunos métodos del tipo `List<T>`. Los compartidos con `Set<T>`, comentados anteriormente, tienen la misma firma en `Set<T>` que en `List<T>`.

```

public interface List<T> ... {
    int size();
    T get(int index);
    T set(int index, T element);
    boolean add(T element );
    void add(int index, T element );
    boolean isEmpty();
    boolean contains(Object o);
    int indexOf(Object o);    ...
}

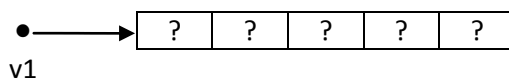
```

Anidamiento de agregados: Arrays, Listas y Conjuntos.

Los elementos que hay en una *array*, lista o conjunto pueden ser otro agregado de datos. Veamos algunos ejemplos.

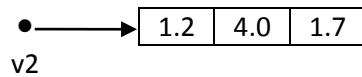
```
int[ ] v1 = new int [5];
```

Define un *array* para 5 valores enteros cuyo valor está indefinido en general aunque para algunos tipos se escogen los valores por defecto del tipo. En el caso de `Integer` el valor por defecto es 0 por lo que en este caso se rellenarían con cero y para un valor de tipo objeto en general `null`. Pero en general es siempre mejor suponer que el valor está indefinido.



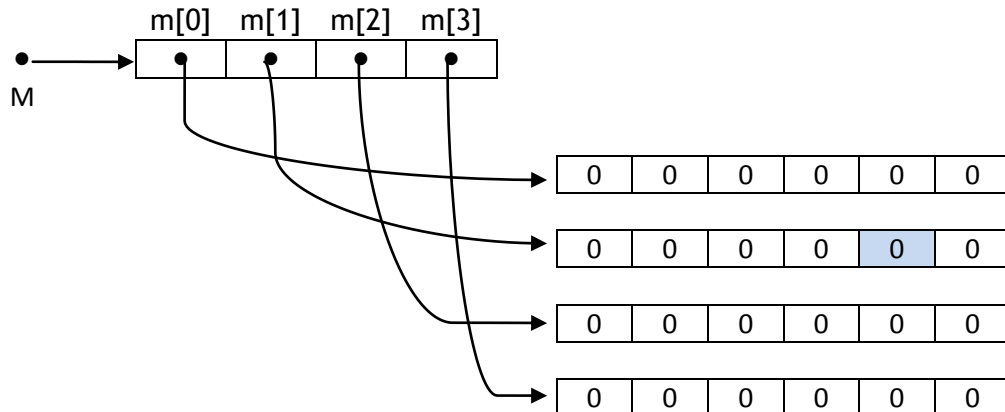
```
Float[ ] v2 = {1.2f, 4.0f, 1.7f};
```

Define un *array* para 3 valores *Float*



```
int [ ][ ] m = new int [4][6];
```

Define un *array* para 4 objetos que a su vez son *arrays* de enteros (**array bidimensional**)



El recuadro sombreado corresponde al elemento `m[1][4]`.

El esquema gráfico puede ser reproducido mediante Listas.

```
List<List <Integer>> v = new ArrayList();
v.add(new ArrayList ());
v.add(new ArrayList ());
v.add(new ArrayList ());
v.add(new ArrayList ());
```

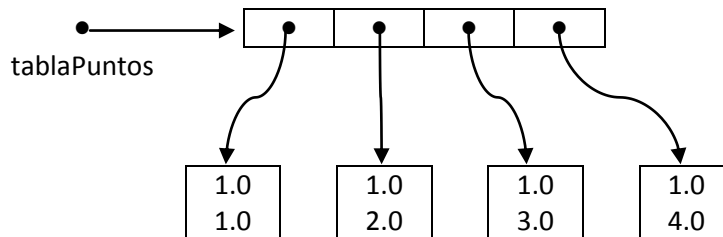
O mediante conjuntos

```
Set<Set<Integer>> c = new HashSet();
c.add(new HashSet());
c.add(new HashSet());
cv.add(new HashSet());
c.add(new HashSet());
```

Tanto en el caso de la lista como del conjunto construimos un agregado de tamaño 4 cada uno de cuyos elementos es un agregado vacío.

```
Punto[ ] tablaPuntos = new Punto[4];
tablaPuntos[0] = new PuntoImpl(1.0,1.0);
tablaPuntos[1] = new PuntoImpl(1.0,2.0);
tablaPuntos[2] = new PuntoImpl(1.0,3.0);
tablaPuntos[3] = new PuntoImpl(1.0,4.0);
```

Define un *array* para 4 objetos *Punto* con los valores devueltos por los constructores respectivos



Tenemos el mismo esquema gráfico pero ahora implementado con listas.

```
List<Punto> tablaPuntosv = new ArrayList();
tablaPuntosv.add(new PuntoImpl(1.0,1.0));
tablaPuntosv.add(new PuntoImpl(1.0,2.0));
tablaPuntosv.add(new PuntoImpl(1.0,3.0));
tablaPuntosv.add(new PuntoImpl(1.0,4.0));
```

O implementado con conjuntos.

```
Set<Punto> tablaPuntosc = new HashSet();
tablaPuntosc.add(new PuntoImpl(1.0,1.0));
tablaPuntosc.add(new PuntoImpl(1.0,2.0));
tablaPuntosc.add(new PuntoImpl(1.0,3.0));
tablaPuntosc.add(new PuntoImpl(1.0,4.0));
```

Otras operaciones con *arrays*

La clase de utilidad [Arrays](http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html) ofrece un conjunto de métodos para realizar operaciones más complejas sobre *arrays* como: buscar un elemento en un *array* (*binarySearch*); saber si dos *arrays* son iguales (*equals*); rellenar un *array* con un elemento dado (*fill*); ordenar un *array* (*sort*); obtener su representación como cadena (*toString*); o devolver una lista con los elementos de un *array* (*asList*). Para más detalles de la clase *Arrays* consulte la documentación de la API de Java¹.

```
package java.util;
public class Arrays {
    public static <T> List<T> asList(T...a);
    //Existen versiones para double, float, char, byte, long y Object de:
    // binarySearch, equals, fill, sort, toString
    public static int binarySearch(int[] a, int key);
    public static int binarySearch(int[] a, int fromIndex, int toIndex, int
key);
    public static boolean equals(int[] a, int[] a2);
    public static void fill(int[] a, int val);
    public static void fill(int[] a, int fromIndex, int toIndex, int val);
    public static void sort(int[] a);
```

¹ <http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html>

```
public static void sort(int[] a, int fromIndex, int toIndex);
public static String toString(int[] a);
...
}
```

4. Expresiones y operadores

En el tema 1 vimos el concepto de expresión.

- Una expresión es un conjunto constantes, variables, operadores.
- Una expresión bien formada tiene siempre un tipo y una vez evaluada devuelve un valor de ese tipo.

Conversiones de tipo en expresiones

Para saber si una expresión está bien formada (no tiene errores sintácticos) tenemos que comprender los mecanismos para calcular el tipo de una expresión. Estos mecanismos incluyen las conversiones de tipos tanto automáticas o implícitas como explícitas.

Los tipos de datos primitivos tienen unas reglas sencillas de conversión. Entre ellos existe un orden: *int*, *long*, *float*, *double*. Este orden nos permite deducir cuándo un tipo primitivo se puede convertir en otro automáticamente.

- Los tipos primitivos se convierten al nivel superior de forma automática cuando es necesario.
- En el sentido inverso las conversiones deben ser explícitas (mediante un operador de casting). En este caso los decimales se truncan cuando pasamos de números reales a enteros.
- La conversión entre tipos de datos primitivos y sus correspondientes envolturas es automática cuando es necesario.
- Las operaciones aritméticas (+, -, *, /) se realizan siempre entre operandos del mismo tipo. Para ello cuando se intenta hacer una operación aritmética entre dos tipos primitivos distintos primero el tipo menor se convierte al mayor en el orden anterior. El tipo devuelto es el mayor. La operación realizada es la correspondiente al tipo mayor. Así 5/2 es de tipo entero y su resultado es 2. Sin embargo 5./2 es de tipo *double* y su resultado es 2.5. El operador % (resto) toma dos operandos de tipo entero y devuelve un tipo entero.

Para comprender las reglas de conversión entre tipos objetos es conveniente partir del grafo de tipos que explicaremos más adelante. Aunque en resumen:

- Un tipo se convierte automáticamente cuando es necesario en cualquier supertipo. Volveremos a aclarar más este concepto cuando veamos, más adelante, el grafo de tipos.
- El operador `new` seguido de un constructor devuelve objetos del tipo de la clase a la que pertenece el constructor. Podemos concluir de las dos reglas anteriores que cuando se crea un objeto su tipo puede ser convertido automáticamente en:
 - El tipo de la clase a la que pertenece.
 - El tipo de cualquiera de las interfaces que implementa su clase o alguna de sus clases padre

- El tipo *de* cualquiera de sus supertipos del tipo asociado a la clase en el grafo de tipos.
- Como veremos más adelante cualquier tipo puede convertirse automáticamente en un tipo proporcionado por *Java* en el tipo *Object*.
- En general el tipo de un objeto puede ser convertido explícitamente (casting) a uno de los tipos que ofrece. Si el tipo al que se quiere convertir no es ofrecido por el objeto se producirá un error en tiempo de ejecución y se lanzará la excepción *ClassCastException*. Los **tipos ofrecidos por un objeto** son todos los supertipos del tipo asociado a la clase que ha creado el objeto.

Operadores de Asignación e Igualdad: Concepto de igualdad e identidad.

Los operadores de asignación e igualdad en Java son respectivamente: `=`, `==`. Para llevar a cabo la operación de asignación se convierte automáticamente el tipo del operando derecho en el tipo del operando izquierdo, según las reglas explicadas arriba, y se devuelve el tipo del operando izquierdo. Si la conversión no es posible la expresión no está bien formada. Tras la ejecución del operador de asignación la variable de la izquierda cambia su valor al devuelto por la expresión de la derecha. Esto hace que la izquierda de un operador de asignación no pueden aparecer constantes.

El operador de igualdad devuelve un tipo *boolean* y toma dos operandos del mismo tipo (para ello es posible que haya que hacer alguna conversión automática).

Estos operadores tienen un significado muy preciso según se apliquen entre tipos primitivos o tipos objeto. Para aclarar ese significado debemos introducir el concepto de identidad y su diferencia con la igualdad.

De manera general, dos objetos son **iguales** cuando los valores de sus propiedades observables son iguales. Por otro lado, dos objetos son **idénticos** cuando al modificar una propiedad observable de uno de ellos, se produce una modificación en la del otro y viceversa. De lo anterior, se deduce que **identidad implica igualdad**, pero igualdad no implica identidad. Es decir la identidad de un objeto permanece inalterada aunque cambien sus propiedades. Dos objetos no idénticos pueden tener las mismas propiedades y entonces son iguales. Dos objetos idénticos siempre tienen las mismas propiedades.

Los valores de los tipos primitivos pueden tener igualdad pero no tienen identidad. El operador `new` crea objetos con una nueva identidad. El operador de asignación entre tipos objeto asigna la identidad del objeto de la derecha al objeto de la izquierda. Después de asignar el objeto *b* al *a* (`a = b;`) ambos objetos son idénticos y por lo tanto si modificamos las propiedades de *a* quedan modificadas las de *b* y al revés. Entre tipos primitivos el operador de asignación asigna valores. Si *m* y *n* son de tipos primitivos entonces `m=n;` asigna el valor de *n* a *m*. Después de la asignación *m* y *n* tienen el mismo valor pero si modificamos el valor de *m* no queda modificado el de *n* porque los elementos de tipos primitivos no tienen identidad sólo tienen valor.

El operador `==` aplicado entre tipos primitivos decide si los valores de los operandos son iguales. Este operador aplicado entre tipos objeto decide si ambos objetos son idénticos o no. Todo objeto *o1* en Java dispone de un método que veremos en el siguiente tema para decidir si es igual a otro objeto *o2* aunque ambos no sean idénticos (es decir tienen las propiedades

indicadas iguales aunque sus identidades sean distintas). Éste es el método `equals` que se invoca mediante la expresión `o1.equals(o2)`; que devuelve un valor tipo *boolean*.

Cuando un objeto es inmutable no pueden cambiarse sus propiedades. Cualquier operación sobre el objeto que cambien las mismas producirá como resultado un nuevo objeto (con una nueva identidad) con los valores adecuados de las propiedades.

Ejemplo (operadores de asignación e igualdad con tipos primitivos):

```
int i = 7;
int j = 4;
int k = j;
boolean a = (i ==j ) ; // a es false
boolean b = (k ==j ) ; // b es true
```

El valor de *a* es *false*. Se comparan los valores de tipos primitivos y estos valores son distintos. Después de asignar *j* a *k* sus valores son iguales luego *b* es *true*.

Ejemplo (igualdad e identidad con objetos):

```
Punto p1 = new PuntoImpl(1.0,1.0);
Punto p2 = new PuntoImpl(1.0,1.0);
boolean c = (p1 == p2) // c es false
boolean d = p1.equals(p2); // d es true
```

Los objetos *p1* y *p2* han sido creados con dos identidades distintas (cada vez que llamamos al operador `new` se genera una nueva identidad) y no han sido asignados entre sí (no son idénticos) pero son iguales porque sus propiedades los son. Por ello *c* es *false* y *d* es *true*.

Ejemplo (identidad de objetos):

```
Punto p1 = new PuntoImpl(1.0,1.0);
Punto p2 = new PuntoImpl(3.0,1.0);
Punto p3 = p1;
p1.setX(3.0);
boolean a = (p3 == p1 ) // a es true
boolean b = (p3 == p2) // b es false
Double x1 = p3.getX() // x vale 3.0
```

El valor de *a* es *true* porque *p1* ha sido asignado a *p3* y por lo tanto tienen la misma identidad. Por ello al modificar la propiedad *X* en *p1* queda modificada en *p3* y por lo tanto *x1* vale 3.0.

Ejemplo (identidad con tipos inmutables):


```
Integer a = 3;
Integer b = a;
b++;
boolean e = (a==b);    // e es false
```

Al ser el tipo *Integer* inmutable el valor de *e* es *false*. En la línea tres (*b++*) se crea un objeto nuevo (con una nueva identidad) que se asigna a *b*. Si se elimina la línea 3 (*b++*;) entonces el valor de *e* es *true*.

Operadores Java

Los operadores son signos especiales –a veces, conjuntos de dos caracteres– que indican determinadas operaciones a realizar con las variables y/o constantes sobre las que actúan en el programa.

Operadores aritméticos

Operadores aritméticos	
+	Suma
-	Resta
*	Producto
/	División
%	Módulo

Los operadores aritméticos toman operandos del mismo tipo y devuelven ese tipo. Si hubiera dos tipos distintos primero se convierte del tipo menor al mayor. Los operandos deben ser de tipos primitivos, si son de tipo envoltura entonces primero se produce una conversión automática al tipo primitivo correspondiente. Los cuatro primeros operadores pueden tomar operandos de cualquier tipo numérico. El operador % solo toma operandos enteros.

Operadores lógicos

Operadores lógicos	
&&	y (and)
	o (or)
!	no (not)

Algunos operadores tienen una evaluación perezosa. Esto quiere decir que solo evalúan los elementos relevantes para obtener el resultado. Los operadores lógicos && y || son de este tipo. Así la expresión *e1* && *e2* evalúa primero *e1* y si da como resultado *false* devuelve el resultado *false* sin evaluar *e2*. Si *e1* es *true* entonces evalúa *e2* y devuelve el resultado. La expresión *e1* || *e2* evalúa en primer lugar *e1*; si da *true* devuelve ese resultado sin evaluar *e2*. Si la evaluación de *e1* es *false* entonces evalúa *e2* y devuelve el resultado. Esta forma de

evaluación hace que estos operadores no sean conmutativos y esto se puede aprovechar para escribir algunas expresiones en una forma que no produzcan errores en tiempo de ejecución. Por ejemplo:

Ejemplo (evaluación perezosa de operadores lógicos)

```
boolean b = (n != 0) && (x < 1.0/n)
```

La expresión en el ejemplo anterior no produce errores en tiempo de ejecución debido a la semántica perezosa del operador &&.

Operadores relacionales

Operadores de relación	
>	mayor que
<	menor que
>=	mayor o igual que
<=	menor o igual que
==	igual que/idéntico a
!=	distinto de/no idéntico a

Si los operandos son tipos primitivos el operador == evalúa si el valor de los operandos es igual o no. Si el tipo de los operandos es un tipo objeto en ese caso este operador evalúa si la identidad de los operandos es la misma o no.

Otros operadores

Otros operadores	
.	Invocar método
(tipo)	Conversión de tipo
[]	Acceso a posición de array
instanceof	Pregunta si un objeto es de un tipo
new	Crear objetos
?:_	Operador condicional ternario

El operador ternario ?: tiene evaluación perezosa. Este operador toma tres operandos: $e0$, $e1$, $e2$ en la forma $e0?:e1:e2$, donde $e0$ debe ser de tipo *boolean*. La semántica del mismo es la siguiente: se evalúa $e0$, si es true evalúa $e1$ y devuelve el resultado (no evalúa $e2$) y si es false evalúa $e2$ y devuelve el resultado (no evalúa $e1$).

Junto a los anteriores también está el operador de asignación (=, ya comentado arriba) y sus formas asociadas +=, -=, etc.

Operadores de asignación

Operadores de asignación	
Abreviado	No abreviado
<code>a += b</code>	<code>a = a + b</code>
<code>a -= b</code>	<code>a = a - b</code>
<code>a *= b</code>	<code>a = a * b</code>
<code>a /= b</code>	<code>a = a / b</code>
<code>a %= b</code>	<code>a = a % b</code>
<code>a++</code>	<code>a = a + 1</code>
<code>a--</code>	<code>a = a - 1</code>

Si el tipo de los operandos en los operadores de asignación es un tipo objeto inmutable como por ejemplo *Integer* entonces la equivalencia de `a+=b` es `a = new Integer(a+b)`. Ocurre lo mismo para los otros operadores y tipos inmutables. Es decir se crea un objeto nuevo (una nueva identidad). Igualmente para esos tipos inmutables la equivalencia de `a++` es `a = new Integer(a+1)`. Siempre debemos tener en cuenta que los operandos de los operadores aritméticos deben ser tipos primitivos.

Orden de prelación de operadores: precedencia y asociatividad

El resultado de una expresión depende del orden en que se ejecutan las operaciones. Por ejemplo, si queremos calcular el valor de la expresión `3 + 4 * 2`, podemos tener distintos resultados dependiendo de qué operación se ejecuta primero. Así, si se realiza primero la suma (`3+4`) y después el producto (`7*2`), el resultado es 14; mientras que si se realiza primero el producto (`4*2`) y luego la suma (`3+8`), el resultado es 11.

Para saber en qué orden se realizan las operaciones es necesario definir unas reglas de precedencia y asociatividad. La tabla siguiente resume las reglas de precedencia y asociatividad de los principales operadores en el lenguaje Java.

Operador	Asociatividad
<code>.</code> <code>[]</code> <code>()</code>	
<code>+</code> <code>-</code> <code>!</code> <code>++</code> <code>--</code> (tipo) <code>new</code>	derecha a izquierda
<code>*</code> <code>/</code> <code>%</code>	izquierda a derecha
<code>+</code> <code>-</code>	izquierda a derecha
<code><</code> <code><=</code> <code>></code> <code>>=</code>	izquierda a derecha
<code>==</code> <code>!=</code>	izquierda a derecha
<code>&&</code>	izquierda a derecha
<code> </code>	izquierda a derecha
<code>?:</code>	
<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code>	derecha a izquierda



Los operadores que están en la misma línea tienen la misma precedencia, y las filas están en orden de precedencia decreciente.

El uso de paréntesis **modifica** el orden de aplicación de los operadores. Si no hay paréntesis el orden de aplicación de los operadores viene definido por la prioridad y la asociatividad de los mismos definida arriba.

Algunos operadores derivados

Junto a los operadores ofrecidos directamente por Java es conveniente disponer de algunos más que usaremos en adelante. Son, entre otros, los **operadores lógicos de implicación y doble implicación o equivalencia**. El operador de implicación lo representaremos de la forma siguiente, donde $e1$ y $e2$ son dos expresiones lógicas:

$$e1 \rightarrow e2 \equiv !e1 \text{ ? true : } e2$$

Una manera sencilla de implementarlo, como se ve arriba, es usando el operador ternario `_?_:_`.

El operador de doble implicación se representa por $e1 == e2$. Es la igualdad entre expresiones que devuelven valores de tipo boolean.

Otras operaciones aritméticas:

La clase [*Math*](#) ofrece un conjunto de métodos adecuados para el cálculo del valor absoluto, máximo, mínimo, potencia, raíz cuadrada, etc. Todos para distintos tipos de datos primitivos.

```
package java.lang;
public class Math{
    public static double abs(double a);
    public static int max(int a, int b);
    public static long min(long a, long b);
    //Existen versiones para double, float, int y long de min, max y abs
    public static double pow(double b, double e);
    public static double random();
    public static double sqrt(double a);
    ...
}
```

- `double abs(double a)`: Valor absoluto
- `int max(int a, int b)`: Máximo de dos números
- `long min(long a, long b)`: Mínimo dos números
- `double pow(double b, double e)`: Potencia
- `double random()`: Número aleatorio entre 0 y 1
- `double sqrt(double a)`: Raíz cuadrada

Otros cálculos con números enteros que pueden ser útiles los ubicamos en la clase `Enteros`. Abajo aparecen métodos para decir si un entero es múltiplo o divisor de otro.

```
public class Enteros {
```

```

    public static boolean esMultiplo(Integer a, Integer b){
        return (a%b) == 0 ;
    }
    public static boolean esDivisor(Integer a, Integer b){
        return (b%a)==0;
    }
    public boolean esPrimo(Integer a){
        ...
    }
    public Integer factorial(Integer a){
        ...
    }
    public Integer mcd(integer a, Integer b){
        ...
    }
    public Integer mcm(Integer a, Integer b){
        ...
    }
    ...

```

Más adelante aprenderemos a diseñar los métodos que no están completos pero supondremos que los tendremos disponibles para ser usados en otros ejercicios. Son métodos que calculan la factorial, si un número es primo, el máximo común divisor, el mínimo común múltiplo, etc.

5. Sentencias de control

El cuerpo de los métodos de una clase está formado por una serie de unidades elementales que llamaremos sentencias. La sentencia más sencilla es una expresión. Ahora introduciremos unas sentencias más complejas como son las sentencias de control. Estas sentencias sirven para romper el orden lineal de ejecución que hemos visto hasta ahora en las secuencias de expresiones.

Un **bloque** está formado por un conjunto de sentencias entre { y }. Dentro de un bloque puede haber declaraciones cuyo ámbito se extiende hasta el final del bloque. Un bloque también es una sentencia.

5.1. Sentencia if-else

Evalúa una condición y según sea cierta o falsa ejecuta un bloque de sentencias u otro.

Sintaxis:

```

if (condición) {
    sentencia-1;

```

```

...
sentencia-n;
} else {
    sentencia-n+1;
    ...
    sentencia-m;
}

```

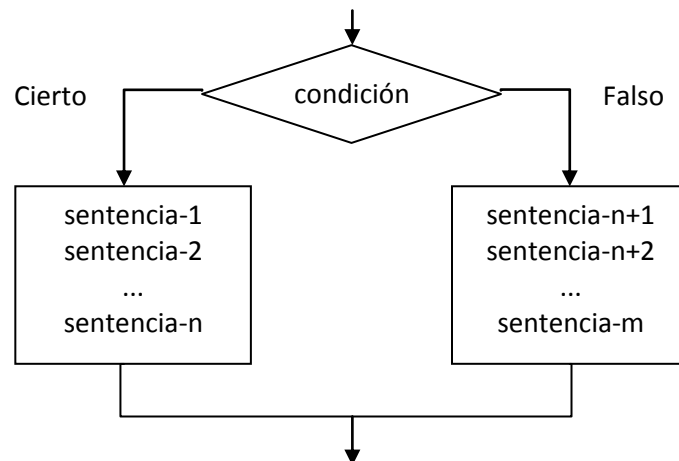


Figura. Esquema de la sentencia if-else

Ejemplo: código que calcula el mínimo de dos enteros.

```

public static Integer minimo(Integer a, Integer b){
    Integer r;
    if(a<=b)
        r = a;
    else
        r = b;
    return r;
}

```

Cómo funciona la sentencia *if*? En primer lugar, se evalúa la expresión lógica que figura como condición. Esta evaluación debe dar un valor de *true* o un valor de *false*. Si el valor de la condición es cierto (*true*) entonces sólo se ejecutarán las sentencias de la 1 a la n. Si, por el contrario, la condición es falsa, se ejecutarán sólo las sentencias de la n+1 a la m.

Algunas cuestiones sintácticas acerca de *if*:

1. La condición debe estar siempre entre paréntesis.
2. La sección *else* con el bloque de las sentencias correspondiente es opcional, es decir, puede que haya un *if* sin sección *else*.

3. Las llaves que marcan los bloques son obligatorias sólo cuando hay más de una sentencia. Sin embargo, se recomienda ponerlas siempre, aun cuando sólo haya una sentencia.
4. Como se puede observar, no hay punto y coma después de la condición del *if*.

Ejemplo (sentencia if):

```
public static void testParidad(Integer n){
    if (n%2==0){
        mostrar ("Es par");
    }
    else {
        mostrar ("Es impar");
    }
}
```

Ejemplo (sentencia if sin bloque else):

```
public static void testFactorial(Integer n){
    if (n>0){
        Double f = Utiles.factorial (n);
        mostrar("El factorial es ", f);
    }
}
```

Ejemplo (sentencia if con bloque else y llaves obligatorias):

```
public static void testTrigonometria(Double x){
    if (x>0 && x<1){
        Double s=Math.sin(x);
        mostrar ("El seno es ", s);
    }
    else {
        Double c=Math.cos(x);
        mostrar ("El coseno es ", c);
    }
}
```

Las sentencias *if-else* pueden encadenarse haciendo que se evalúe un conjunto de condiciones y se ejecute una sola de las opciones entre varias.

Ejemplo (if-else encadenados):

```
public static Float calcularImpuesto(Double salario){
    Float impuesto = 0.0;
    if (salario>=5000.0){
        impuesto = 20.0;
    }
    else if (salario<5000.0 && salario>=2500.0){
        impuesto = 15.0;
    }
    else if (salario<2500.0 && salario >=1500.0){
        impuesto = 10.0;
    }
    else if (salario > 800.0){
        impuesto = 5.0;
    }
}
```

```

    }
    return impuesto;
}

```

Otra posibilidad es que en los bloques de sentencias correspondientes al *if* o al *else* haya otros *if*. A esta estructura se le llama *if-else* anidados. Por ejemplo, para detectar el cuadrante en el que se encuentran las coordenadas *x* e *y* de un punto dado, podemos escribir el siguiente método de test:

Ejemplo (if-else anidados):

```

public static void testCuadrante(Punto p) {
    if (p.getX() >= 0.0) {
        if (p.getY() >= 0.0) {
            mostrar ("Primer cuadrante");
        }
        else {
            mostrar ("Cuarto cuadrante");
        }
    }
    else {
        if (p.getY() >= 0.0) {
            mostrar ("Segundo cuadrante");
        }
        else {
            mostrar ("Tercer cuadrante");
        }
    }
}

```

Las sentencias con *if* anidados pueden presentar problemas de legibilidad si no empleamos las llaves de bloque convenientemente. Por ejemplo,

Ejemplo (if anidados sin llaves)

```

if (a >= b)
    if (b != 0.0)
        c = a/b;
    else
        c = 0.0;

```

la sentencia anterior puede parecer ambigua porque no se han puesto llaves, y podrían surgir dudas respecto a si el *else* corresponde al primer o al segundo *if*. La clave no está en la indentación (el compilador de Java se saltará todos los espacios en blanco y tabuladores al compilar) sino en la regla de que un *else* siempre corresponde al *if* anterior. Si se quiere modificar esta regla es obligatorio usar llaves:

Ejemplo (if con llaves necesarias)


```

if (a >= b) {
    if (b != 0.0)
        c = a/b;
} else
    c = 0.0;

```

Aunque, como hemos señalado antes, un buen programador siempre pone todas las llaves, aunque no hagan falta:

Ejemplo (if con todas las llaves)

```

if (a >= b) {
    if (b != 0.0) {
        c = a/b;
    }
} else {
    c = 0.0;
}

```

5.2. La sentencia switch

Cuando se requiere comparar una variable de un tipo discreto con una serie de valores diferentes, puede utilizarse la sentencia *switch*, en la que se indican los posibles valores que puede tomar la variable y las sentencias que se tienen que ejecutar si la variable coincide con alguno de dichos valores. Es una sentencia muy indicada para comparar una variable de un tipo enumerado con cada uno de sus posibles valores. Este tipo de programas siempre pueden hacerse con sentencias *if-else* pero cuando se trata de comparar una variable que toma valores discretos con cada uno de ellos y el número de los mismos es 3, 4 ó superior entonces es preferible, por legibilidad del programa y eficiencia, una sentencia *switch*. Su sintaxis es:

```

switch( variable ){
    case valor1: sentencias; break;
    case valor2: sentencias; break;
    ...
    case valorN: sentencias; break;
    default:     sentencias;
}

```

Se ejecutan las sentencias del *case* cuyo valor es igual al de la variable. Si el valor de la variable no coincide con ningún valor, entonces se ejecutan las sentencias definidas en *default*, si es que las hay.

Es interesante destacar la sentencia *break* que va detrás de cada *case*. Aunque como la sentencia *break* es opcional, es necesario ponerlas porque sin ellas la sentencias *case* fallarían, en el sentido de que sin un *break*, el flujo del programa seguiría secuencialmente a través de todas las sentencias *case*. Veamos esta cuestión comprobando cómo funciona la sentencia *switch* más detalladamente.

Supongamos, en primer lugar, que no hay ninguna sentencia *break*. En ese caso, se evalúa *expresión* que debe devolver obligatoriamente un valor de los tipos indicados anteriormente. Si este valor coincide con el valor constante *valor_1*, se ejecutan todos los bloques de sentencias desde la 1 hasta la n. Si el resultado coincide con el valor constante *valor_2*, se ejecutan los bloques desde el 2 hasta el n. En general, si el valor de *expresión* es igual a *valor_i*, se ejecutan todos los bloques desde i hasta n. Si ningún *valor_i* coincide con el valor de *expresión* se ejecutará, si existe, el *bloque_sentencia_n+1* que está a continuación de *default*.

Si se desea ejecutar únicamente un *bloque_sentencia_i* hay que poner una sentencia *break* a continuación. El efecto de la sentencia *break* es dar por terminada la ejecución de la sentencia *switch*.

Ejemplo (sentencia case)

```
public static String getDiaSemana(Integer dia){
    String s;
    switch(dia){
        case 1:
            s = "Lunes";
            break;
        case 2:
            s = "Martes";
            break;
        case 3:
            s = "Miércoles";
            break;
        case 4:
            s = "Jueves";
            break;
        case 5:
            s = "Viernes";
            break;
        case 6:
            s = "Sábado";
            break;
        case 7:
            s = "Domingo";
            break;
        default:
            s = "Error";
    }
    return s;
}
```

Existe también posibilidad de ejecutar el mismo *bloque_sentencia_i* para varios valores del resultado de expresión. Para ello se deben poner varios *case expresion_cte :* seguidos.

Ejemplo (sentencia switch con el mismo bloque para varios case)

```
public static Integer getNumDiasMes(Integer mes, Integer anyo) {
```

```

Integer res= null;
switch (mes) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        res = 31;
        break;
    case 4: case 6: case 9: case 11:
        res = 30;
        break;
    case 2:
        if (Fechas.esBisiesto(anyo)) {
            res = 29;
        } else {
            res = 28;
        }
}
return res;
}

```

5.3. Sentencia while

Ejecuta el bloque de sentencias mientras la condición evalúa a cierta.

Sintaxis:

```

while (condición) {
    sentencia-1;
    sentencia-2;
    ...
    sentencia-n;
}

```

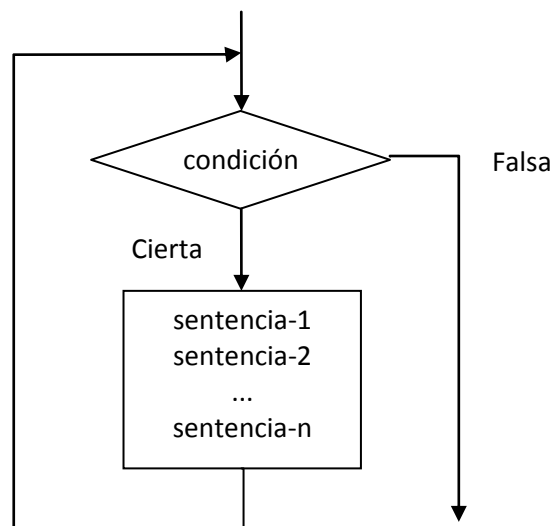


Figura. Esquema de la sentencia while

El funcionamiento de la sentencia *while* es el siguiente: Primero se evalúa la condición, que debe ser una expresión booleana. Si es cierta, se ejecutan las sentencias de la 1 a la n y, una vez acabadas, se vuelve a evaluar la condición. Si sigue siendo cierta, se ejecuta de nuevo el bloque de sentencias, y así sucesivamente, hasta que la condición sea falsa.

Algunas cuestiones a remarcar sobre la sentencia *while*:

1. Al igual que en la sentencia *if*, la condición debe ir entre paréntesis.
2. Normalmente una sentencia *while* tiene más de una sentencia en su bloque asociado, por eso las llaves son obligatorias.
3. Entre las sentencias del bloque debe haber alguna que modifique el valor de alguna de las variables que intervienen en la condición, para que en algún momento, ésta se evalúe a falso.
4. El bucle *while* no tiene expresamente una inicialización, así que es tarea del programador inicializar convenientemente las variables que intervienen en el bucle.
5. Si la primera vez que se evalúa la condición, fuera falsa, las sentencias del bloque no se ejecutarían ni una sola vez.

Ejemplo (sumar los n primeros números naturales usando while)

```
public static Integer sumatorioWhile(Integer n){
    Integer s = 0;
    int i = 1;
    while(i<=n){
        s = s + i;
        i++;
    }
    return s;
}
```

5.4. Sentencia for clásico

Se empieza ejecutando la inicialización (INI) y evaluando la condición (CON), a continuación, y mientras la condición se evalúe como true se ejecuta el bloque de sentencias y la actualización (ACT). Sintaxis:

```
for (INI; CON; ACT) {
    sentencia-1;
    sentencia-2;
    ...
    sentencia-n;
}
```

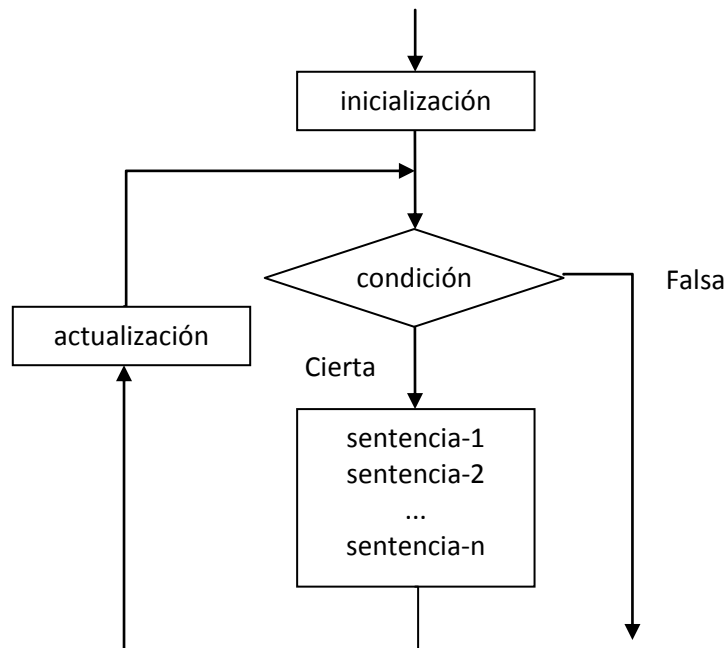


Figura. Esquema de la sentencia for clásico

¿Cómo funciona el *for*? Cuando el control de un programa llega a una sentencia *for*, lo primero que se ejecuta es la *inicialización*. A continuación, se evalúa la *condición*, si es cierta (valor distinto de cero), se ejecuta el bloque de sentencias de 1 a n. Una vez terminado el bloque, se ejecuta la sentencia de actualización antes de volver a evaluar la condición, si ésta fuera cierta de nuevo se ejecutaría el bloque y la actualización, y así repetidas veces hasta que la condición fuera falsa y abandonáramos el bucle.

Algunos detalles de la sintaxis:

1. Como se puede observar, las tres partes de la sentencia *for* están entre paréntesis y separadas por un punto y coma, pero no se escribe un punto y coma después del paréntesis cerrado.
2. Si el bloque de sentencias a ejecutar está compuesto por sólo una sentencia, no es obligatorio poner llaves, aunque, como señalamos antes, es muy ponerlas siempre por legibilidad.
3. Si después de la inicialización, la condición fuera falsa, las sentencias del bloque no se ejecutarían ni una sola vez.

Ejemplo (sumar los n primeros números naturales con for)

```

public static Integer sumatorioFor(Integer n){
    Integer s = 0;
    for(int i=0; i <=n; i++){
        s = s + i;
    }
    return s;
}
  
```

Ejemplo (sumar los números de un array de enteros con for clásico) :

```
public static Integer sumarForClasico(Integer[] t){
    Integer s = 0;
    for(int i=0; i <=t.length; i++){
        s = s + t[i];
    }
    return s;
}
```

Sentencia while/for clásico: ejemplos

Método que devuelve la suma de los números naturales desde 1 hasta el valor de n

```
public static int suma(int n){
    int i = 1;
    int s = 0;
    while(i <=n){
        s = s + i;
        i++;
    }
    return s;
}
```

El mismo método anterior realizado con for clásico sería

```
public static int suma(int n){
    int s = 0;
    for(int i= 1;i <= n; i++){
        s = s + i;
    }
    return s;
}
```

5.5. Sentencia for-each o for extendido

En Java se define un agregado como un conjunto de datos que se pueden recorrer en un determinado orden. Un agregado puede ser un *array*, una Lista un conjunto u otros tipos de agregados, que veremos más adelante, cuyos tipos sean subtipos del tipo *Iterable<T>* (que se verá más adelante). En la sentencia for each se ejecuta el bloque de sentencias mientras quedan elementos por recorrer en el agregado .

Sintaxis:

```
for (tipo variable : agregado) {
    sentencia-1;
    sentencia-2;
    ...
    sentencia-n;
}
```

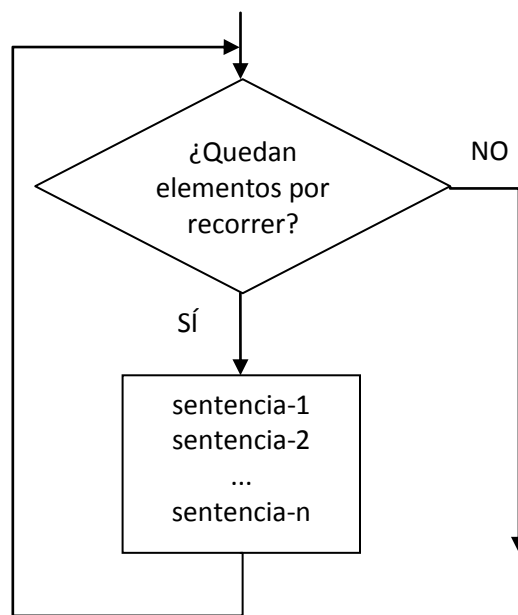


Figura. Esquema de la sentencia for extendido

Ejemplo (sumar los números de un array de enteros con for extendido) :

```

public static Integer sumarForExtendido(Integer[] t){
    Integer s = 0;
    for(Integer num: t){
        s = s + num;
    }
    return s;
}

```

Sentencias for clásico / for each: ejemplos

Código que calcula en la variable *s* la suma de los números contenidos en el *array* *t* mediante un *for* clásico.

```

int [] t = {1,3,5,7,11};
s = 0;
for (int i= 0; i < t.length; i++) {
    s = s + t[i];
}

```

Código que calcula en la variable *s* la suma de los números contenidos en el *array* *t* mediante un *for*-each

```

int [] t = {1,3,5,7,11};
s = 0;
for (int e : t) {
    s = s + e;
}

```

```
}
```

Equivalencias entre *for* extendido, *for* clásico y *while*

Todas las estructuras repetitivas son equivalentes entre sí. Se recomienda utilizar el *for* extendido. Es menos probable cometer errores con él cuando se es un programador novel.

Abajo se muestran las equivalencias entre la estructura construida con *for* extendido, *for* clásico y *while*. En cada una de ellas la variable *c* es de tipo *array* de *T*.

for extendido	for clásico	while
<pre>T[] c =; for (T e: c) { sentencia-1; ... sentencia-n; }</pre>	<pre>T[] c =; for (int i=0; i<c.length; i++) { T e = c[i]; sentencia-1; ... sentencia-n; }</pre>	<pre>T[] c = ...; int i = 0; while (i<c.length) { T e = c[i]; sentencia-1; ... sentencia-n; i++; }</pre>

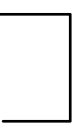
5.6. Sentencias *continue* y *break*

Las sentencias **continue** y **break** son aplicables en bloques de programa que están dentro de estructuras iterativas. Con ellas se pretende que mediante la evaluación de una condición se continúe evaluando el bloque o termine la iteración.

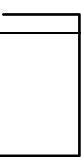
La sentencia **continue** dentro de un bloque de sentencias que forma parte de una estructura iterativa hace que el flujo de programa se salte sin evaluar el resto del bloque, continuando en la evaluación de la condición con la siguiente iteración.

La sentencia **break** dentro de un bloque de sentencias que forma parte de una estructura iterativa hace que el flujo de programa salte fuera del bloque, dando por finalizada la ejecución de la sentencia iterativa.

```
for/while ( ← ) {
    sentencia-1;
    ...
    if (...) continue;
    ...
    sentencia-n;
}
```



```
for/while (    ) {
    sentencia-1;
    ...
    if (...) break;
```




```

    ...
    sentencia-n;
}

```

6. Relaciones entre interfaces y clases: herencia, implementación y uso. Grafo de tipos

La **herencia** es una propiedad por la que se establecen relaciones que podemos denominar **padre-hijo entre interfaces o entre clases**. La herencia se representa mediante la cláusula **extends**.

En Java la herencia entre interfaces puede ser múltiple: un hijo puede tener uno o varios padres. La interfaz hija tiene todas las signaturas (métodos) declaradas en los padres. No se puede ocultar ninguna.

La sintaxis para la **interfaz** con herencia responde al siguiente patrón:

```

[Modificadores] interface NombreInterfazHija extends NombreInterfacesPadres
[,...] {
    [...]
    [signatura de métodos]
}

```

Mediante **extends** se indica que la interfaz que estamos definiendo combina otras interfaces padres y, posiblemente, añade algunos métodos más. Cuando una interfaz extiende a otra decimos que el tipo definido por la interface hija es un **subtipo** del tipo definido por la interfaz padre.

Una clase puede heredar de otra clase. Esto se expresa mediante la cláusula **extends**. En Java una clase solo puede heredar de una clase padre. Una clase puede implementar varios interfaces. Esto se especifica mediante la cláusula **implements**.

La sintaxis para la **clase** hija responde al siguiente patrón:

```

[Modificadores] class NombreClaseHija extends NombreClasePadre
                                [implements NombreInterfaz, ... ] {
    [atributos ]
    [métodos ]
}

```

Mediante **extends** se indica que la clase que estamos definiendo hereda de otra clase padre. La relación de herencia entre clases indica que todos los métodos públicos (junto con su código) están disponibles para la clase hija. Entre los métodos públicos se incluyen los constructores públicos. Más adelante veremos que también están disponibles los métodos y atributos con otras visibilidades.

En la clase hija se puede acceder a los métodos de la clase padre mediante la palabra reservada **super**. Con esta palabra reservada se pueden formar nuevas expresiones en el cuerpo de una clase que hereda de otra. Las expresiones que se pueden formar se componen de la palabra **super** y el operador punto (.) seguido de algunos de los métodos de la clase padre visibles en la clase hija. Por ahora, los únicos métodos visibles son los públicos (**public**). La palabra reservada **super** también sirve para acceder a los constructores de la clase padre visibles desde la hija. En este caso **super** irá seguido de paréntesis y una secuencia de parámetros, y representará la invocación del correspondiente constructor de la clase padre. Es decir, el constructor de la clase padre, visible desde la hija y que tenga los parámetros que se han proporcionado la tras la palabra **super**. Algunos ejemplos de expresiones de este tipo son:

Ejemplo (usos de la palabra reservada **super**):

```
super.m(2.0);
.....
super(2,3);
```

En la primera se invoca al método *m* de la clase padre con parámetro 2.0. Y en la segunda, se invoca al constructor de la clase padre que tendrá dos parámetros de tipo *int*.

Una clase sólo puede heredar de otra clase, pero puede implementar varias interfaces. La relación que se establece entre una clase y cada una de las interfaces que implementa la llamaremos **relación de implementación**. La relación de implementación obliga a la clase a implementar (proporcionar código) a cada uno de los métodos de la interfaz implementada.

Al igual que con interfaces, cuando una clase hereda de otra, el tipo definido por la clase hija es un **subtipo** del tipo definido por la clase padre. Entre tipos y subtipo hay una importante propiedad. Sea una variable *b* de un tipo *B*, y otra *a* de otro tipo *A*. Si *B* es un subtipo de *A*, entonces la variable *b* puede ser asignada a la variable *a* pero no al revés.

```
A a;
B b;
a = b;    // correcto
b = a;    // incorrecto
```

Ejemplos concretos son:

- *A* es una interfaz y *B* otra que hereda de él.
- *A* es una clase y *B* otra que hereda de ella.
- *A* es una interfaz y *B* una clase que la implementa.

Ejemplo (herencia de interfaces: **Pixel**, **Vector2D**):

```
public enum Color {
```

```

    ROJO, NARANJA, AMARILLO, VERDE, AZUL, VIOLETA
}
public interface Pixel extends Punto {
    Color getColor();
    void setColor(Color c);
}
public interface Vector2D extends Punto {
    void diferencia(Vector2D v);
    void suma (Vector2D v);
    void multiplicaPor(Double d);
    Double getProductoEscalar(Vector2D v);
    Double getProductoCruzado(Vector2D v);
    Double getModulo();
}

```

La interfaz *Pixel* hereda de *Punto*, y añade una nueva propiedad, consultable y modificable, denominada *Color*. La interfaz *Vector2D* hereda de *Punto* y añade las propiedades *productoEscalar*, *productoCruzado* y *modulo*. Además, añade otras operaciones como *diferencia*, *suma* y *multiplicaPor*.

Una vez que hemos diseñado los tipos anteriores, debemos implementarlos. La implementación del tipo *Pixel* la hacemos en la clase *PixelImpl*. Para implementar esta clase reutilizamos la clase *PuntoImpl*, heredando de ella. La implementación de la clase *Vector2DImpl* la dejamos como ejercicio, teniendo en cuenta que dados dos vectores, en dos dimensiones, $(x_1, y_1), (x_2, y_2)$, las diferentes propiedades y operaciones sobre ellos vienen definidas por:

- Suma: $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$
- Diferencia: $(x_1, y_1) - (x_2, y_2) = (x_1 - x_2, y_1 - y_2)$
- MultiplicaPor: $k(x_1, y_1) = (kx_1, ky_1)$
- ProductoEscalar: $(x_1, y_1) * (x_2, y_2) = (x_1x_2 + y_1y_2)$
- ProductoCruzado: $(x_1, y_1) \times (x_2, y_2) = (x_1y_2 - y_1x_2)$
- Modulo: $|(x_1, y_1)| = \sqrt{x_1x_1 + y_1y_1}$

Ejemplo (implementación de *PixelImpl* reutilizando mediante Herencia *PuntoImpl*)

```

public class PixelImpl extends PuntoImpl implements Pixel {
    private Color color;
    public PixelImpl () {
        super();
        this.color = Color.VERDE;
    }
    public PixelImpl (Double x, Double y, Color color) {
        super(x, y);
        this.color = color;
    }
    public Color getColor () {
        return this.color;
    }
    public void setColor (Color color) {
        this.color = color;
    }
    public String toString() {

```

```

        String s = super.toString();
        s = s+"."+color;
        return s;
    }
}

```

Como podemos ver, dentro del cuerpo de la clase *PixelImpl* se usan las palabras reservadas *super* y *this*. La clase *PixelImpl* debe implementar *Pixel*, es decir, todos sus métodos, pero para ello reutiliza el código ya disponible en *PuntoImpl*. Dentro del código de *PixelImpl*, que hereda de *PuntoImpl*, *super.m(..)* se refiere al método *m* de la clase *PuntoImpl*. Igualmente, *super(x,y)* se refiere al constructor con dos parámetros *Double* de la clase *PuntoImpl*. Sin embargo, *this.color* se refiere al atributo *color* de la clase *PixelImpl*.

La clase *Vector2DImpl* tendrá la cabecera

```
public class Vector2DImpl extends PuntoImpl implements Vector2D {...}
```

Grafo de tipos

El conjunto de tipos que hemos definido en los ejemplos anteriores, y las relaciones entre ellos son, se muestran en la figura.

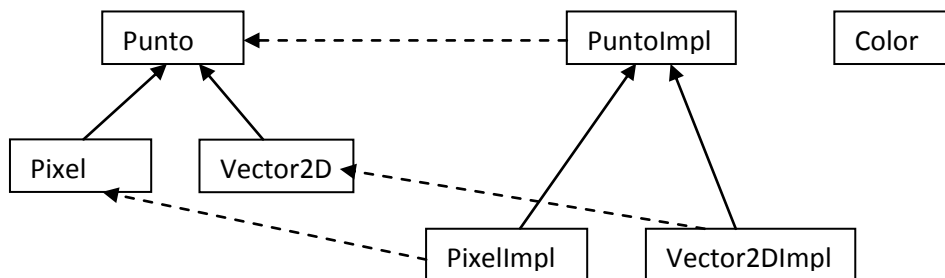


Figura. Grafo de tipos.

A este grafo lo llamaremos **grafo de tipos**. En este grafo representamos los tipos y sus relaciones. Los tipos que aparecen definidos en el grafo de tipos son tipos definidos por una interfaz, por una clase o por una declaración *enum*. En el grafo hay representadas tres tipos de relaciones:

- Herencia de clases.
- Herencia de interfaces.
- Implementación de un interfaz por una clase.

Hemos representado las relaciones de herencia (de clases o interfaces) con flechas continuas y las de implementación con flechas discontinuas.

El grafo de tipos nos permite visualizar rápidamente los tipos que hemos diseñado y su relación de subtipado. De una forma general, un tipo es subtipo de otro si hay un camino del primero al segundo a través de flechas de herencia o de implementación en el grafo de tipos. Si un tipo B es un subtipo de A, entonces decimos que A es un **supertipo** de B. El grafo de tipos

es útil, también, para visualizar el conjunto de **tipos ofrecidos por un objeto**. El conjunto de tipos ofrecidos por un objeto está formado por todos los supertipos del tipo asociado a la clase que creó el objeto. Así, si creamos un objeto de la clase *PixellImpl*, éste ofrece los tipos: *PixellImpl*, *Pixel*, *Punto*, *PuntoImpl*. En este conjunto siempre se añade un tipo proporcionado por Java y que es ofrecido por todos los objetos: el tipo *Object*.

Junto a las anteriores relaciones entre los tipos hay una más que no se suele representar en el grafo de tipos. Es la **relación de uso**. Un tipo *A* usa a otro *B* cuando declara algún parámetro formal, tipo de retorno, atributo o variable local del tipo *B*. Así por ejemplo, el tipo *Pixel* usa *Color* pero el tipo *Punto* no.

7. Gestión de excepciones

Las excepciones son, junto con las sentencias de control vistas anteriormente, otro mecanismo de control. Es decir, es un instrumento para romper y gestionar el orden en que se evalúan las sentencias de un programa.

Se denomina **excepción** a un evento que ocurre durante la ejecución de un programa, y que indica una situación normal o anormal que hay que gestionar. Por ejemplo, una división por cero o el acceso a un fichero no disponible en el disco. Estos eventos pueden ser de dos grandes tipos: eventos generados por el propio sistema y eventos generados por el programador. En ambos casos hay que gestionar el evento. Es decir, decidir qué sentencias se ejecutan cuando el evento ocurre.

Cuando se produce un evento como los comentados arriba decimos que se ha **disparado una excepción**. Cuando tomamos decisiones después de haberse producido un evento decimos que **gestionamos la excepción**.

Hasta ahora hemos visto los métodos de los diferentes tipos como mecanismos adecuados para llevar a cabo una acción pasándole unos parámetros y posiblemente obteniendo un resultado. Ahora un método puede generar excepciones que habrá que gestionar si alguna de ellas se dispara durante su llamada. Hay una segunda clasificación de las excepciones que se pueden generar dentro de un método: las que estamos obligados a declarar en la signature del mismo y las que no. Vemos, por lo tanto, que la signature de un método incluirá además del nombre, los parámetros formales y el tipo devuelto las excepciones que pueden generarse dentro del método y que estamos obligados a declarar.

Cuando se dispara una excepción tras la ocurrencia de un evento se crean objetos que transportan la información del evento. A estos objetos también los llamamos excepciones. En cada programa necesitamos excepciones (objetos) de tipos específicos. Esto lo conseguimos diseñando clases para este fin. Algunas vienen ya predefinidas y otras tenemos que diseñarlas. A este proceso lo llamamos **diseño de excepciones**. Veamos con más detalle cada uno de los aspectos de la programación con excepciones.

Diseño de Excepciones

Es el mecanismo de diseño de las nuevas clases que nos permitirán crear los objetos que se crearán cuando se dispara una excepción. El entorno Java ya tiene un conjunto de excepciones predefinidas y una jerarquía de las mismas. La figura muestra la jerarquía de Excepciones, incluyendo algunas de las más notables de la API de Java.

Las excepciones heredan del tipo *Throwable*, aunque en este texto sólo prestaremos atención a las que heredan del tipo *Exception*. Para nosotros, por lo tanto, los objetos que se crean al dispararse una excepción serán de un tipo hijo de *Exception*.

Hay un subtipo específico de *Exception* que se denomina *RuntimeException*. Si una excepción hereda de *RuntimeException* no es obligatorio declararla en la signatura aunque pueda ser generada en el cuerpo del método. Una excepción que se puede generar en el cuerpo de un método pero que no extienda *RuntimeException*, aunque sí a *Exception*, tiene que ser declarada en la signatura obligatoriamente. Esta característica es muy importante a la hora de diseñar las excepciones: al diseñar una nueva excepción y por tanto una nueva clase, debemos decidir si hereda de *RuntimeException* o no, esto es, directamente de *Exception*. Esta decisión influirá sobre las signaturas de los métodos y el código de las llamadas a los mismos como veremos más adelante.

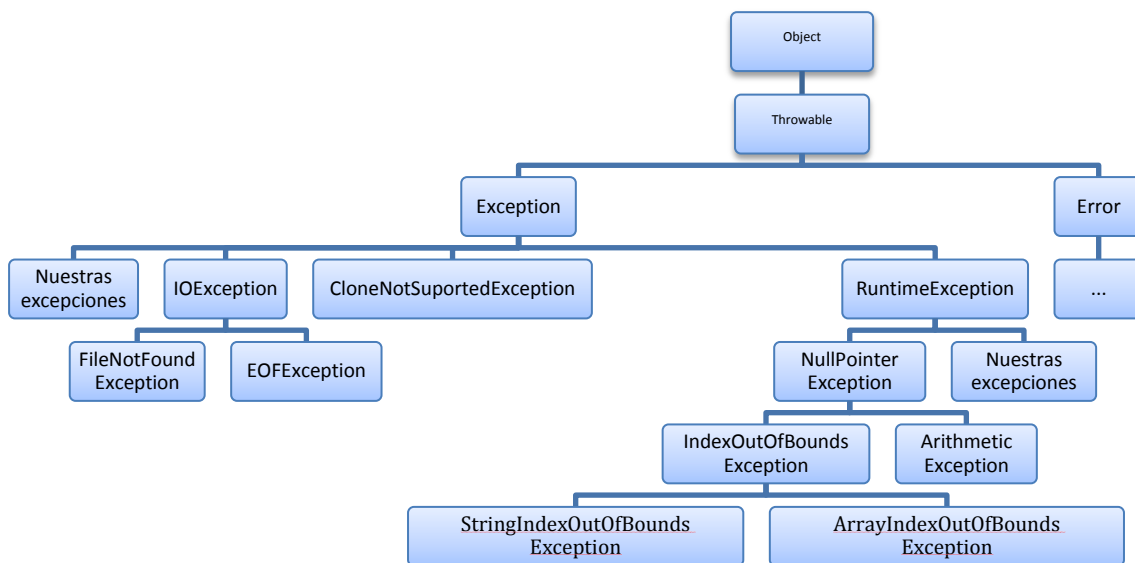


Figura. Jerarquía de excepciones de la API de Java

Ejemplo

```

public class FueraDeRangoException extends Exception {
    public FueraDeRangoException() {
        super();
    }
    public FueraDeRangoException(String s) {
        super(s);
    }
}

```

Se ha diseñado una nueva excepción llamada *FueraDeRangoException* que hereda de *Exception* con dos constructores: uno sin parámetros y otro con un *String* que puede recoger detalles del evento particular ocurrido. Siempre recurrimos a los constructores de la clase padre mediante la invocación al constructor *super(..)*.

Declaración de excepciones en las signatures de los métodos: la cláusula throws

Cuando un método puede generar excepciones dentro de su cuerpo y estas excepciones son hijas de *Exception* pero no de *RuntimeException* tienen que declararse en su signature mediante la cláusula *throws*. Esta declaración debe hacerse en interfaces y clases.

La sintaxis para declarar un método en las interfaces queda ahora:

```
tipo nombre-método (parámetros-formales) throws ClaseException 1, ClaseException2, ...;
```

y en las clases:

```
tipo nombre-método (parámetros-formales) throws ClaseException1, ClaseException, ... {  
    cuerpo  
}
```

Ejemplo:

```
public FileReader(String fileName) throws FileNotFoundException;
```

El constructor *FileReader* intenta abrir un fichero de nombre *filename*. Si no lo encuentra disparará la excepción *FileNotFoundException* que es una excepción predefinida hija de *Exception*.

Disparo de excepciones: la cláusula throw

Cuando se dispara una excepción el flujo se interrumpirá en ese punto y se propagará la excepción a los métodos invocantes hasta que sea gestionada o finalice el programa.

El disparo de una excepción suele tener la siguiente estructura:

```
if ( condicion_de_disparo ) throw new MiExcepcion("TextoExplicativo");
```

Como vemos es una sentencia *if* que evalúa una condición y si es cierta dispara una excepción. Llamaremos **Condición de Disparo** a esa condición que si es cierta disparará la excepción. Cada excepción tiene asociado un evento de disparo. El evento ocurre cuando la Condición de Disparo se hace verdadera. La Condición de Disparo es una expresión lógica que depende de los parámetros del método y de las propiedades del objeto. La cláusula *throw MiExcepcion("TextoExplicativo")* crea un objeto del tipo adecuado y dispara la excepción. Por cada posible excepción a disparar debemos escoger una Condición de Disparo (en general un evento) que cuando se hace verdadera se dispare la excepción.

Vemos que hay dos cláusulas: *throw* y *throws*. La primera sirve para disparar excepciones. La segunda para declararlas en las signatures de los métodos.

Las excepciones disparadas por sistema (por ejemplo las que pueden aparecer en operaciones aritméticas o las que se disparan cuando se intenta acceder a una celda no permitida de un Vector o un array y que provienen del hardware), tienen un funcionamiento parecido.

Gestión de excepciones: las cláusulas `try`, `catch`, `finally`

En general un método tiene dos modos de funcionamiento que dependen de los parámetros reales que reciba y las propiedades del objeto sobre el que se invoca el método. En el **modo normal** el método termina sin disparar excepciones y devolviendo el resultado adecuado. En el **modo excepcional** el método dispara una excepción y termina. Hay una tercera posibilidad y es que el método no termine. Si eso ocurre hay un problema de diseño en el código del método.

Cuando un método *m1* llama a otro *m2* entonces *m1* puede gestionar las excepciones que *m2* pueda generar o alternatively propagarlas al método que llamó a *m1*. Es decir no gestionarlas. Si *m2* declara en su signature la posibilidad de generar excepciones mediante la cláusula *throws* entonces *m1* tiene la obligación, en su código, de gestionar todas las excepciones declaradas.

Las excepciones no gestionadas se propagan desde cada método al que lo ha llamado hasta que se encuentra un método que las gestione. Si ningún método gestiona una excepción, ni tampoco el programa principal, entonces el programa termina de forma abrupta.

Si el método *m1* llama a *m2* entonces la gestión de las excepciones generadas por *m2* se hace en el código de *m1* mediante las cláusulas *try/catch/finally*. El código que debe contener *m1* para gestionar las posibles excepciones disparadas por *m2* es:

```
try {
    Sentencias0;
} catch (TipoDeException1 e1) {
    Sentencias1;
} catch (TipoDeException2 e2) {
    Sentencias2;
} finally {
    Sentenciasf;
}
```

Las sentencias contenidas en el bloque del *try*, *Sentencias0*, deben contener la llamada al método *m2*. El bloque *try* es el código donde se prevé que se genere una excepción. Es como si dijésemos "intenta estas sentencias y mira a ver si se produce una excepción".

Si en la llamada al método *m2*, dentro de *Sentencias0*, se produce una excepción entonces *m2* termina y pasa el control a *m1*. El método *m1* va comprobando de forma secuencial si la excepción disparada es la contemplada en cada una de las cláusulas *catch* que siguen al bloque *try*. Si es así se ejecuta el bloque de sentencias asociado a esa cláusula *catch*. Las sentencias en el bloque *finally* se ejecutan siempre al final se haya disparado excepción o no.

Resumiendo podemos decir que la llamada al método que puede disparar excepciones se hace dentro del bloque del *try*. Si el método llamado funciona en modo normal termina y posteriormente se ejecutan las sentencias del bloque *finally*. Si el método funciona de modo excepcional y hay un bloque *catch* para gestionar la excepción disparada entonces se ejecutan

las sentencias del bloque *catch* y después las del bloque *finally*. Si el método funciona en modo excepcional y no hay ningún bloque *catch* para gestionar la excepción disparada se ejecutan las sentencias del bloque *finally* y se propaga la excepción al método que llamó a *m1*.

El bloque *finally* es opcional. También son opcionales las cláusulas *catch*. Pero siempre debe haber una cláusula *finally* o al menos una cláusula *catch* detrás del bloque *try*.

Ejemplos de gestión de excepciones y funcionamiento de las mismas

```
public class TestExcepcion1 extends Test {
    public static void main(String[] args) {
        Integer n = 1/0;
        mostrar(n);
    }
}
```

En el ejemplo anterior no se gestiona la excepción que aparece al intentar la división por cero. El programa termina con el mensaje:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at TestExcepcion1.main(TestExcepcion1.java:8)
```

En el ejemplo siguiente se gestiona la excepción.

```
public class TestExcepcion2 extends Test {
    public static void main(String[] args) {
        Integer n = null ;
        try {
            n = 1/0;
        }
        catch (ArithmeticException e1) {
            mostrar (n);
            mostrar ("El valor es infinito ");
        }
        mostrar ("Final del programa");
    }
}
```

El resultado que aparece es:

```
El objeto es: null
El valor es infinito
Final del programa
```

En el bloque *try* se ha producido una excepción al intentar dividir por cero. Como no se ha podido hacer la división no se ha asignado ningún valor a *n* cuyo valor sigue siendo *null*. La excepción disparada es *ArithmeticException*. Como existe un bloque *catch* para gestionar esta excepción se ejecuta el código correspondiente, después se ejecutará el código posterior a las cláusulas *try/catch*.

En los ejemplos siguientes se diseña una nueva excepción hija de *RuntimeException*, se muestra cómo se dispara y qué ocurre si se gestiona o no.

```
class DivisorCero extends RuntimeException {
    public DivisorCero( ) {
        super();
    }
    public DivisorCero(String texto ) {
        super(texto);
    }
}
```

En el ejemplo siguiente se dispara la excepción diseñada previamente si el parámetro *b* es igual a cero. La excepción no tiene que ser declarada en la signatura del método, en este caso un constructor, mediante la cláusula *throws* porque la excepción hereda de *RuntimeException*.

```
public RacionalImpl(Integer a, Integer b) {
    if(b==0) {
        throw new DivisorCero("El divisor no puede ser cero");
    }else{
        num=a;
        den=b;
    }
}
```

```
public class TestExcepcion3 extends Test {
    public static void main(String[] args) {
        Racional r = null ;
        r = new RacionalImpl(1,0);
        mostrar (r);
        mostrar (" Final del programa");
    }
}
```

En el ejemplo anterior no se gestiona la excepción. El programa termina apareciendo el mensaje:

```
Exception in thread "main" Temal.DivisorCero: El divisor no puede ser cero
at Temal.RacionalImpl.<init>(RacionalImpl.java:8)
at Temal.TestExcepcion.main(TestExcepcion.java:6)
```

```
public class TestExcepcion4 extends Test {
    public static void main(String[] args) {
        Racional r = null ;
        try {
            r = new RacionalImpl(1,0);
            mostrar (r);
        }
        catch (DivisorCero e1) {
            mostrar(e1.getMessage());
        }
        mostrar (r);
    }
}
```

```

        mostrar ("Final del programa");
    }
}

```

En el ejemplo anterior sí se gestiona la excepción. Tras el disparo de la excepción en la llamada al constructor *RacionalImpl* se ejecutan las sentencias del bloque catch que gestiona la excepción *DivisorCero*. El resultado mostrado es:

```

El divisor no puede ser cero
El objeto es: null
Final del programa

```

8. Semántica del paso de parámetros en Java

El mecanismo de paso de parámetros varía de unos lenguajes a otros. Recordamos del tema 1 que los **parámetros formales** son variables que aparecen en la signatura del método en el momento de su declaración. Los **parámetros reales** son expresiones que se colocan en el lugar de los parámetros formales en el momento de la llamada al método.

En la llamada a un método se asignan los parámetros reales a los formales, se ejecuta el cuerpo del método llamado y se devuelve el resultado al llamador. En este mecanismo los parámetros podemos dividirlos en dos grupos: **parámetros de entrada** y **parámetros de entrada-salida**. Cuando llamamos a un método le pasamos unos parámetros reales. La diferencia entre los tipos de parámetros radica en el hecho que los parámetros reales pasados al método puedan ser cambiados de valor o no por las operaciones llevadas a cabo dentro del cuerpo del mismo. Si no pueden cambiar son parámetros de entrada y si pueden cambiar parámetros de entrada salida. Ambos tipos de parámetros pueden ser distinguidos según el tipo del parámetro formal correspondiente.

Un parámetro es de entrada si el tipo de parámetro formal es un tipo primitivo o un tipo objeto inmutable. Un parámetro es de entrada-salida si el tipo del parámetro formal es un tipo objeto mutable. Veamos algunos ejemplos.

Supongamos el siguiente método *m* (en este caso *static*) que es llamado desde el método *main*:

```

public class TestParametros extends Test {

    public static double m(Integer i, String s, double d, Punto p) {
        i++;
        s += " Modificado";
        d = i;
        p.setX(d);
        return d;
    }
}

```

```

    }

    public static void main(String[] args) {
        Integer i1 = 14;
        String s1 = "Valor Inicial";
        double d1 = 0.75;
        Punto p1 = new PuntoImpl(3.0,4.5);
        double r = m(i1,s1,d1,p1);
        mostrar("i1 = "+i1);
        mostrar("s1 = "+s1);
        mostrar("d1 = "+r);
        mostrar("p1 = "+ p1);
    }
}

```

El resultado obtenido es:

```

i1 = 14
s1 = "Valor Inicial";
d1 = 0.75
p1 = (15.0,4.5)

```

En el ejemplo anterior los parámetros formales son *i*, *s*, *d*, *p* cuyos tipos son *Integer*, *String*, *double*, *Punto*. Es decir y por este orden, dos objetos inmutables, un tipo primitivo y un objeto mutable. Como podemos observar por el resultado las operaciones en el cuerpo del método *m* han modificado el parámetro real *p1* (tipo objeto mutable) pero no los parámetros reales *i1*, *s1*, *d1* (tipos inmutables, tipo primitivo). Es decir los parámetros *i*, *s*, *d* se han comportado como parámetros de entrada y *p* como parámetro de entrada-salida. Esto ocurre en general: *los tipos inmutables y los tipos primitivos se comportan como parámetros de entrada y los tipos objeto mutables como parámetros de entrada-salida*. Pero ¿por qué ocurre así? Para entenderlo vamos a transformar el programa en otro equivalente que nos permita comprender las propiedades del paso de parámetros a partir de las propiedades de la asignación.

La transformación consiste en sustituir la llamada por un nuevo bloque de código. En ese bloque por cada parámetro formal se declara una variable del mismo tipo y con el mismo identificador, se asignan a esas variables los valores de los parámetros reales, se ejecuta el código y al final se asigna la expresión en el *return* a la variable que recogía el valor de llamada del método llamador.

El programa anterior es equivalente al siguiente:

```

public class TestParametros2 extends Test {
    public static void main(String[] args) {
        Integer i1 = 14;
        String s1 = "Valor Inicial";
        double d1 = 0.75;
        Punto p1 = new PuntoImpl(3.0,4.5);
        double r;
        {
            Integer i;

```

```

        String s;
        double d;
        Punto p;
        i = i1;
        s = s1;
        d = d1;
        p = p1;
        i++;
        s+= " Modificado";
        d = i;
        p.setX(d);
        r = d;
    }
    mostrar("i1 = "+i1);
    mostrar("s1 = "+s1);
    mostrar("d1 = "+r);
    mostrar("p1 = "+ p1);
}

```

Veamos en primer lugar los tipos primitivos. En estos la asignación cambia los valores del operando de la derecha al de la izquierda. Pero estos tipos no tienen identidad. Por lo tanto el parámetro formal *d* recibe el valor del correspondiente parámetro real *d1* pero las modificaciones en el valor del parámetro formal *d* no afectan al parámetro real *d1*. Es un parámetro de entrada.

En los tipos objetos inmutables (*Integer*, *String*, ...) la asignación asigna la identidad de los operados de la derecha a los operandos de la izquierda. En las primeras líneas *i*, *s* tienen la misma identidad que *i1*, *s1*. Pero las operaciones sobre estos tipos (*i++*, *s+= " Modificado"*) generan una nueva identidad que queda guardada en *i*, *s*. Por lo tanto los cambios en los parámetros formales no afectan a los parámetros reales. Como en el caso de los tipos primitivos son parámetros de entrada.

En el caso de los parámetros de tipo objeto mutable los parámetros formales y los reales comparten la misma identidad, tras la asignación. Los cambios en los parámetros formales afectan a los parámetros reales. Por tanto, son siempre parámetros de entrada-salida.

Número variable de parámetros: operador *varargs*

El operador *varargs*, que se denota por "...", puesto detrás del tipo de un parámetro formal indica que el método aceptará una secuencia variable de parámetros del mismo tipo.

El tratamiento de los parámetros de este tipo se hará como si el parámetro formal fuera una array del tipo correspondiente. Es decir si *T* es un tipo, la expresión *T...* es equivalente en su tratamiento a *T[]*. Sin embargo la forma de aportar los parámetros reales es distinta. Si el parámetro formal es *T[]* el parámetro real debe ser un array de *T*. Pero si el parámetro formal es *T...* los parámetros reales son una secuencia de expresiones de tipo *T* separadas por comas.

```

public class TestSaludos extends Test{
    public static void imprimeSaludos(String... names) {
        for (String n : names) {
            mostrar("Hola " + n + ". ");
        }
    }
}

```

```
public static void main(String[] args) {
    imprimeSaludos("Pablo", "Antonio", "Juan");
}
```

El resultado es

```
Hola Pablo.
Hola Antonio.
Hola Juan.
```

El operador *varargs* (...) siempre se coloca al final en la signatura del método y se debe evitar su uso con métodos sobrecargados.

9. Tipos genéricos y métodos genéricos

En Java existe la posibilidad de usar **tipos genéricos**. Estos son tipos que dependen de parámetros formales que serán instanciados posteriormente por tipos concretos. En Java las interfaces, las clases y los métodos pueden ser genéricos. Esto quiere decir que tienen parámetros en su definición que posteriormente se instanciarán por tipos concretos. En este caso decimos que son interfaces, clases o métodos genéricos. Se denomina **Programación Genérica** a la programación que usa tipos, interfaces, clases y métodos genéricos.

La programación genérica permite **programas más reutilizables y con mayor tolerancia a fallos**.

Tipos, Interfaces, clases y métodos genéricos

Como ejemplo de tipo genérico tenemos el *array* de un tipo *T*.

```
T[] a;
```

Con la declaración anterior decimos que *a* es un array cuyos elementos son de tipo *T*.

Como primer ejemplo de interfaz genérica tenemos la interfaz *Comparable* que estudiaremos con más detalle en el tema siguiente:

```
interface Comparable<T> {
    int compareTo(T e);
}
```

Esta interfaz solo tiene un método, *compareTo*, cuyo objetivo es comparar el objeto *this* con el objeto *e* y devolver 0, un número negativo o positivo según que *this* sea igual, menor o mayor que *e*, respectivamente. Vemos que la interfaz genérica *Comparable* tiene un parámetro formal *T*. Esto se declara escribiendo *<T>* después del nombre de la interfaz. Ese parámetro, una vez declarado, puede ser usado para construir tipos genéricos, declarar tipos de parámetros formales o tipos devueltos por métodos. Puede haber más de un parámetro formal. En este caso la declaración de los parámetros genéricos se hace incluyéndolos

separados por comas entre <>. La declaración de los parámetros genéricos siempre va detrás del nombre de la interfaz.

Una interfaz genérica define un nuevo tipo genérico que puede ser usado en declaraciones posteriores. En este caso el tipo genérico definido es *Comparable<T>*.

El segundo ejemplo es el tipo *List<T>* que vimos al principio del tema. Este tipo es genérico:

```
public interface List<T> ... {

    int size();
    T get(int index);
    T set(int index, T element );
    boolean add(T element );
    void add(int index, T element );
    boolean isEmpty();
    boolean contains(Object o);
    int indexOf(Object o);
    ...
}
```

La declaración de los parámetros genéricos, como en la interfaz, va detrás del nombre de la clase. Como vemos el parámetro genérico es *T* y se ha usado para declarar tipos de parámetros formales y tipos devueltos por los métodos. En una clase también se puede usar para declarar, de forma genérica, el tipo de un atributo. Las clases genéricas, como las interfaces, declaran un nuevo tipo genérico. En este caso el tipo *Vector<T>*.

En una clase no genérica puede haber métodos genéricos. En este caso la declaración de los parámetros genéricos va delante del tipo devuelto por el método y detrás de los modificadores del mismo.

```
public static <T> List<T> nCopias(int n, T a){
    List<T> v = new ArrayList<T>();
    for(int i=0; i<n; i++){
        v.add(a);
    }
    return v;
}
```

En el ejemplo anterior aparece un método genérico de una clase no genérica (Utiles). Este método devuelve una lista con elementos de tipo *T* que contiene *n* copias idénticas del objeto que recibe como parámetro.

Instanciación de tipos, interfaces, clases y métodos genéricos.

Los tipos genéricos, interfaces, clases y métodos deben instanciarse antes de ser usados. Es decir, los parámetros genéricos deben sustituirse por parámetros concretos.

```
public class TestGenerico extends Test{
    public static void main(String[] args) {
        Integer a = 14;
        List<Integer> v;
```

```

        v = Utiles.nCopias(10,a);
        mostrar(v);
    }
}

```

En el ejemplo se ha instanciado el tipo genérico *List<T>* a *List<Integer>*. Igualmente se ha instanciado el método *nCopias*. Ahora la instanciación ha sido deducida por el compilador. El mecanismo por el cual el compilador induce la instanciación necesaria se denomina **inferencia de tipos**. Java tiene esta característica. Si fuera necesario explicitar la instanciación del método pondríamos *Utiles.<Integer>nCopias(10,a)*.

Los tipos genéricos en tiempo de ejecución

NOTA: este apartado contiene información más detallada y se deja como lectura recomendada al alumno que desee ampliar sus conocimientos en este tema.

Sobre los tipos genéricos podemos hacer una operación que llamaremos **operación de borrado** (en inglés *erasure*). Esta operación toma un tipo genérico y devuelve el **tipo desnudo** (*raw type*, es más descriptivo tipo desnudo que la acepción más literal tipo crudo) correspondiente. Un tipo desnudo es no genérico. La operación de borrado de un tipo genérico elimina todo lo que va entre *<>* y devuelve un tipo desnudo, es decir sin *<_>* detrás del nombre del tipo. Por ejemplo la operación borrado de *List<T>* devuelve *List* un tipo desnudo correspondiente a *List<T>*.

Debemos tener en cuenta que los tipos genéricos, en Java, sólo existen en tiempo de compilación. En tiempo de ejecución sólo existen los tipos desnudos correspondientes al tipo genérico. Esto implica que determinados operadores que tienen un tipo como operando sólo admitirán tipos no genéricos. Es el caso del operador *instanceof*. Este operador sólo admitirá como segundo operando un tipo concreto o el tipo desnudo de un tipo genérico. Este comportamiento de los tipos genéricos en Java será muy importante más adelante.

El hecho de que los tipos genéricos no existan en Java en tiempo de ejecución, restringe también la posibilidad de constructores de un tipo *T* que ha sido declarado como un parámetro genérico, al igual que ocurre con los inicializadores de array. Si un tipo de la forma *T[]* (array de *T*) no puedan ser instanciado en tiempo de compilación no se permiten constructores e inicializadores del mismo. Esto es debido a que el compilador no tiene los detalles para construir los objetos del tipo genérico. Dan errores de compilación sentencias del tipo dos, tres, cuatro y cinco siguientes:

```

1. a instanceof List                // bien
2. a instanceof List<T>             // mal
3. a instanceof T                   // mal
4. E a = new E();                   // mal
5. E[] = new E[10];                // mal
6. E a = (E) new Object();          // warning

```

La quinta produce un *warning* en tiempo de compilación. Es decir el compilador no garantiza que no se dispare una excepción en tiempo de ejecución cuando se intente hacer el *casting* de la sentencia sexta. Por lo tanto, los *casting* con tipos genéricos darán *warnings* de este tipo.

Otra implicación del modo de funcionamiento de los tipos genéricos en tiempo de ejecución es que el tipo de los parámetros formales que queda en tiempo de ejecución, es sólo el tipo desnudo del tipo genérico. Dos métodos son indistinguibles si después de aplicar la operación de borrado a los tipos genéricos de los parámetros formales resulta el mismo método. En ese caso no se podrán declarar los dos juntos en la misma interfaz o clase. Las dos declaraciones siguientes declaran el mismo tipo y por lo tanto no pueden hacerse juntas en la misma interfaz o clase.

```
static <T> void swap(List<T> v, int i, int j);
static void swap(List<?> v, int i, int j);
```

Tipos Genéricos y Subtipos

Hemos visto que cuando un tipo *T1* es subtipo de otro *T2* (ver grafo de tipos) podemos asignar un objeto de tipo *T1* a otro de tipo *T2*. Igualmente ocurre si el parámetro formal es de tipo *T2* y el real de tipo *T1*. La mezcla de subtipos y tipos genéricos tiene sus particularidades.

En general cuando un tipo *A<T>* es instanciado por dos tipos concretos *T1* y *T2*, tal que *T1* es subtipo de *T2*, los tipos resultantes *A<T1>*, *A<T2>* no son subtipos entre sí. Por tanto, un objeto de tipo *A<T1>* no podría ser asignado a *A<T2>*, ni un parámetro real de tipo *A<T1>* puede sustituir a un parámetro formal de tipo *A<T2>*. La razón de fondo no la vamos a explicar aquí pero se puede encontrar en los tutoriales de Java.

Como ejemplo concreto *List<PuntoImpl>* no es un subtipo de *List<Punto>*. Con las implicaciones anteriores.

10. Conceptos Aprendidos

- Identificadores, palabras reservadas, literales
- Tipos primitivos y tipos envoltura
- Inmutabilidad
- Constantes y variables
- Tipos array, Vector y String. Declaración, inicialización, acceso y funcionalidades
- Conversión de tipo en las expresiones
- Igualdad e identidad
- Operadores
- Sentencias de control: if-else, for clásico, for extendido, while, continue, break. Equivalencias entre for clásico, for extendido y while
- Excepciones. Diseño y gestión. Cláusulas throws, throw, try, catch, finally
- Paso de parámetros en Java. Parámetros formales y reales. Parámetros de entrada y de entrada-salida
- Operador varargs
- Tipos genéricos y programación genérica

11. Ejercicios

1. Diga si los siguientes identificadores son válidos en Java:
 - num
 - num80
 - 80num
 - num_2
 - _num
 - \$pred\$
 - Var
 - If
2. Enumere qué tipos de datos predefinidos, primitivos o de envoltura, conoce para almacenar:
 - Un número entero
 - Un número real
 - Un carácter
 - Un lógico
3. Señale el tipo de los siguientes literales:
 - "Abc"
 - 50
 - null
 - true
 - 'x'
 - 50.5
 - 1L
 - 0.1f
4. Declare variables para almacenar cada uno de los valores anteriores e inicialícelas con los mismos. ¿Qué añadiría a la declaración para que se convirtiesen en constantes?
5. ¿Cuáles de las siguientes expresiones son correctas en Java? Considere declaradas las siguientes variables:

```
int i = 10, j=20;      Float f=0.5f;      Double d=1.0;      char c='d';
String s = "abc";      Boolean b=true;      final Character c2='h';
```

- (i+j) < d
- (i+j) < c
- (i+j) != 5.0f
- (b == i) + (j != 5.0f)
- c!=s
- s+=(s+s)
- (b != (c>c2)) || ((f-d)==(i-j))
- f++;
- (j%=10) == 0

- `c2=c`
6. ¿Cuáles de las siguientes expresiones son correctas en Java? Considere declaradas las siguientes variables:

```
int i=10, j=20; Float f=0.5f; Double d=1.0; char c='d';
String s="abc"; Boolean b=true; final Character c2='h';
```

- `i+j < d`
 - `i+j < c`
 - `i+j != 5.0f`
 - `b == i + j != 5.0f`
 - `s+=s+s)`
 - `b != (c>c2) || (f-d) == (i-j)`
 - `j%=10 == 0`
7. Indique el tipo de cada una de las expresiones de los dos ejercicios anteriores.
8. Señale el valor que devuelve cada una de las expresiones de los ejercicios anteriores.
9. Declara una variable llamada fila de tipo array de Integer. Crea el objeto que puede almacenar 100 enteros.
10. Declara una variable llamada columna de tipo array de Double. Crea el objeto que puede almacenar 100 reales.
11. Declara una variable llamada matriz de tipo array de array de Character. Crea el objeto de este tipo que pueda almacenar 8x10 caracteres.
12. Declara un array llamado primos e inicialízalo para que contenga los diez primeros números primos (1, 2, 3, 5, 7, 11, 13, 17, 19, 23).
13. Declara una matriz llamada pares e inicialízala con 3 filas de 4 columnas cada una, todas ellas con números pares.
14. Declara e inicializa dos variables de tipo cadena de texto. Declara e inicializa una variable lógica que indique si las dos cadenas anteriores son iguales.
15. Crea una nueva cadena de texto como resultado de pasar a mayúsculas alguna de las dos anteriores.
16. Crea una nueva cadena de texto como resultado de eliminar los espacios en blanco de la anterior.
17. Declara e inicializa una variable lógica que indique si dos de las cadenas anteriores tienen el mismo tamaño.
18. Crea una nueva cadena de texto como concatenación de todas las anteriores.
19. Declara e inicializa una variable lógica que indique si alguna de las cadenas anteriores contiene el carácter 'C'.
20. Crea una nueva cadena de texto como resultado de sustituir el carácter 's' por el carácter '\$' en alguna de las cadenas anteriores.
21. Crea una lista de enteros llamada v y añade en los 10 primeros números impares.
22. Declara e inicializa una variable lógica que indique si la lista anterior contiene el elemento 0.
23. Declara e inicializa una variable entera con el valor del 6º elemento de la lista anterior.
24. Declara e inicializa una variable entera con el índice de la primera aparición del valor 3 en la lista anterior.
25. Declara e inicializa una variable de tipo conjunto de número reales
26. Añade al conjunto anterior los números reales que van de 1. a 3. Con incrementos de 0.5
27. Declara e inicializa una variable lógica que indique si el conjunto anterior contiene el elemento 2.5.

28. Declara e inicializa una variable entera con el valor del tamaño del conjunto anterior.
29. Suponiendo el tipo genérico *Pair* con la propiedades:
 - Tipo genérico *Pair* con parámetros *T1*, *T2*
 Propiedades
P1: *T1*, consultable y modificable
P2: *T2*, consultable y modificable
 Escriba una interfaz genérica para definir el tipo.
30. Escribe la clase genérica *PairImpl* que implemente *Pair* y añada, además el método *toString()* que permita convertir un objeto de ese tipo en cadena de caracteres.
31. Cree la clase de utilidad *Utiles* y en ella implemente un método genérico con nombre *elementoEn*. El método tiene un parámetro genérico *T* y dos parámetros formales. El primero es una *List<T>* y el segundo de tipo *int*. El tipo devuelto es *T*. El método devolverá el elemento con el índice dado por el segundo parámetro. El método debe comprobar si el valor del segundo parámetro es un índice adecuado de la lista y si no lo es disparar la excepción *IllegalArgumentException*.
32. Para el siguiente código Java, señale las palabras reservadas, los identificadores, los tipos de datos, las variables, los operadores, las expresiones y las asignaciones que vea.

```
public class Ejemplo extends Test{
    public static void main(void){
        int x=10, y=5;
        char c='a';
        Double f=-95.99;
        String cadena="Texto de ejemplo";
        int z=x+y*20;
        z+=10;
        cadena=cadena+"\n "+z;
    }}

```

33. Indique el valor de la variable "cadena" justo antes de acabar de ejecutarse el código del método *main*.
34. En el siguiente bloque de código identifique, si las hay, las variables inútiles y el código inútil. Supóngase que la variable *a*, de tipo *int*, está declarada en un ámbito superior.

```
{
    int x = 10;  int y;  int c;

    x= x+2;

    y = 14;

    x = y+ 5;

    a = x;

    x = c + 2;

}
```

Suponiendo que no tenemos en cuenta las inicializaciones por defecto hay alguna variable que se esté usando antes de haber sido inicializada?.

35. Implementa los métodos de la clase *Enteros* vista en el capítulo sabiendo que:

- Un número entero es primo si no es divisible por ningún número entero comprendido entre 2 y su raíz cuadrada. Compruebe en el problema que el parámetro proporcionado es un número entero positivo. Dispare la excepción *IllegalArgumentException* si no se cumple esa condición
 - El máximo común divisor de dos números enteros cumple las siguientes propiedades: $mcd(a,0) == a$, $mcd(a,b) == mcd(b,a \% b)$.
 - El mínimo común múltiplo de dos enteros cumple la propiedad: $mcd(a,b) * mcm(a,b) == a * b$ siempre que a y b sean distintos de cero.
 - Implemente un método *static* en la clase *Enteros* que imprima en pantalla los enteros que van desde a hasta b con incremento de c.
 - Implemente un método *static* en la clase *Reales* que imprima en pantalla los números reales que van desde a hasta b con incremento de c.
36. Escriba la clase *TestEjercicio1* que hereda de *Test* con un método *main* que
- a) Cree dos conjuntos vacíos de *Integer*, añada al primero 7, 3, 9, 1, 5, 3 y al segundo 1, 2, 3, 4 y los muestre por pantalla los conjuntos contruidos.
 - b) Muestre el cardinal de cada uno de los conjuntos.
 - c) Pregunte si uno de los conjunto contiene un entero dado.
 - d) Añada un entero al primer conjunto, compruebe el valor devuelto por la operación dependiendo de si el entero estaba ya en el conjunto o no y muestre el conjunto resultante.
 - e) Elimine un entero de uno de los conjuntos. Muestre el resultado.
 - f) Calcule la unión, la intersección y la diferencia de los conjuntos y muestre los resultados.
37. Escriba la clase *TestEjercicio2* con un método *main* que
- a) Cree dos listas vacías de *Character*, añada a la primera 'S', 'E', 'M', 'A', 'N', 'A' y a la segunda 'R', 'A', 'M', 'O' y las muestre por pantalla.
 - b) Muestre el cardinal de cada una de las listas.
 - c) Pregunte si una de las listas contiene una letra dada.
 - d) Pregunte en qué posición está una letra que sí está en una lista y otra que no está.
 - e) Añada al la primera lista 'S', compruebe el valor devuelto por la operación y muestre la lista resultante.