1. Estructuras de datos básicas en Java

1.1. Literales - Tipos de datos primitivos

En Java toda variable **declarada** debe tener su tipo, y además, antes de poder emplearla se debe inicializar a un valor (**definida**). Caso contrario, el compilador indicará el error y no generará los archivos .class. Los tipos primitivos pueden ser:

1.1.1. Enteros

Almacenan como su propio nombre indica números enteros, sin parte decimal. Hay cuatro tipos de enteros:

Tipo	Tamaño (bytes)	Rango de representación	Ejemplo
long	8	-9223372036854775808 a 9223372036854775808	long v1 = 5L;
int	4	-2147483648 a 2147483647	int v2 = 1;
short	2	-32768 a 32767	short v3 = 0;
byte	1	-128 a 127	byte v4 = 5;

1.1.2. Reales

Almacenan números reales, es decir números con parte fraccionaria. Hay dos tipos:

Tipo	Tamaño (bytes)	Rango	Ejemplo
float	4	+3.40282347E+38	float v5 = 4.5F
double	8	+17976931348623157OE+308	double v6 = 0;

Para indicar que una constante es flotante: ej: 2.3 se debe escribir: 2.3F, sino por defecto será double

ATENCION ¡! Dado que Java es un lenguaje ***case sensitive*** (sensible a las mayúsculas/minúsculas), y particularmente en el caso de los tipos de datos, no es lo mismo declarar un dato double que un Double. Las variables declaradas con **tipo** en minúscula (long, float, etc), son datos primitivos del lenguaje, y las declaradas con **tipos** que comienzan con mayúscula (Long, Float, etc), son instancias de clases de Java, conocidas como ***Wrapper*** (envoltura), que proveen diversos métodos para manipular los datos. Las versiones más nuevas de Java hacen transparente para el desarrollador este manejo, dado que desde Java 5 existe una característica denominada Auto-Boxing, mediante la cual el compilador hace automáticamente los pasos para realizar la conversión entre una y otra opción. Para más información recurrir a la documentación de JAVA en http://java.sun.com/javase/6/docs/api/ (paquete java.lang. Number)

Una de las características de las clases ***Wrapper*** es que proporciona constantes que representan el mayor y menor número representables por cada uno de los tipos de datos.

long	int	float	Doublé
Long.MIN_VALUE	Integer.MIN_VALUE	Float.MIN_VALUE	Double.MIN_VALUE
Long.MAX_VALUE	Integer.MAX_VALUE	Float.MAX_VALUE	Double.MAX_VALUE

Ejemplo:

```
double elMaximo = Double.MAX_VALUE;
int    elMinimo = Integer.MIN_VALUE;
```

1.1.3. Caracteres

En Java hay un único tipo de carácter: **char**. Cada carácter en Java esta codificado en formato Unicode. En este formato cada carácter ocupa dos bytes, frente a la codificación en ASCII, dónde cada carácter ocupa un solo byte.
Los char se representan entre comillas simples: char ch = ‘a’, mientras que las cadenas de caracteres usan comillas dobles.

1.1.4. Boolean

Se trata de un tipo de dato que solo puede tomar dos valores: “**true**” y “**false**”. Es un tipo de dato bastante útil a la hora de realizar chequeos sobre condiciones.

1.2. VARIABLES

- **Declaración** implica informarle al compilador de la existencia de una variable (o función), pero sin reservar memoria (caso de las variables) ni conocer como ejecutarla (caso de funciones).
- **Definición** implica que el compilador reserve memoria (caso de las variables) y que se provea en forma completa el cuerpo de la función (caso de las funciones). Está estrechamente relacionada con la **inicialización**, que implica la asignación del valor que se guarda en el espacio "reservado" para esa variable.

La sintaxis de declaración es la siguiente: int i;
Se puede declarar y definir una variable en una misma línea: int i = 0;
Declaración y definición pueden hacerse en líneas diferentes: int i ;
i = 0;
Es posible declarar varias variables en una línea: int i, j, k;

Después de cada línea de código, bien sea de iniciación o de código, va un ; (punto y coma).
Los caracteres aceptados en el nombre de una variable son los comprendidos entre "A-Z", "az", _, $ y cualquier carácter que sea una letra en algún idioma.

1.3. CONVERSIÓN ENTRE TIPOS NUMÉRICOS

Las normas de conversión entre tipos numéricos son las habituales en un lenguaje de programación: si en una operación se involucran varios datos numéricos de distintos tipos, todos ellos se convierten al tipo de dato que permite una mayor precisión y rango de representación numérica; así, por ejemplo:

- Si cualquier operando es double todos se convertirán en double.
- Si cualquier operando es float y no hay ningún double todos se convertirán a float.
- Si cualquier operando es long y no hay datos reales todos se convertirán a long.

Del mismo modo estas normas se extienden para int, short y byte.

La "jerarquía" en las conversiones de mayor a menor es:
double ← float ← long ← int ← short ← byte

Se deben tener en cuenta estas conversiones a la hora de mirar en qué tipo de variable se guarda el resultado de la operación; ésta debe ser, al menos, de una jerarquía mayor o igual a la jerarquía de la máxima variable involucrada en la operación.
Es posible convertir un dato de jerarquía "superior" a uno con jerarquía "inferior", arriesgando la perdida de información en el cambio. Este tipo de operación (almacenar el contenido de una variable de jerarquía superior en una de jerarquía inferior) se denomina **cast**.

```
public class Ejemplo1 {
  public static void main(String[] args) {
        int k, i = 9;
        float j = 47.9F;     // si no se coloca la "F" da error al compilar
        System.out.println("i: "+i + " j: " +j);
        k = (int)j;          // empleo de un cast
        System.out.println("j: " + j + " k: " +k);
        j = k;               // no necesita cast
        System.out.println("j: " + j + " k: " +k);
  }
}
```

Salida obtenida

```
i: 9        j: 47.9
j: 47.9     k: 47
j: 47.0     k: 47
```

1.4. OPERADORES

Los operadores básicos de Java son + , - , * , / para suma, resta, producto y división. El operador / representa la división de enteros si ambos operandos son enteros. Su módulo puede obtenerse mediante el operador %.
Además existen los operadores decremento e incremento: -- y ++ respectivamente. La operación que realizan son incrementar y decrementar en una unidad a la variable a la que se aplican. Su acción es distinta según se apliquen antes (++a) o después (a++) de la variable. El siguiente programa ilustra estos distintos efectos:

```
public class Ejemplo2{
  public static void main(String[] args) {
     int i = 1;
     System.out.println ("i : " + i);
     System.out.println ("++i : " + ++i);     // Pre-incremento. Primero incrementa y luego imprime i
     System.out.println ("i++ : " + i++);     // Post-incremento. Primero imprime "2" y luego incrementa i
     System.out.println ("i : " + i);          //  i por lo tanto vale 3
     System.out.println ("--i : " + --i);       // Pre-decremento. Primero decrementa i y luego lo imprime
     System.out.println ("i-- : " + i--);      // Post-decremento. Primero imprime i y luego decrementa.
     System.out.println ("i : " + i);          // Ahora i vale 1
  }
}
```

Salida obtenida

```
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```

1.5. Operadores especiales

En Java a diferencia de otros lenguajes no existe el operador exponenciación. Tampoco existen los operadores Seno o Coseno. Si se desea realizar una operación de exponenciación se deberá invocar el método correspondiente de la clase Math de Java.lang. Estos métodos son estáticos, por lo que no es necesario crear un objeto de dicha clase. Su sintaxis general es: Math.metodo(argumentos);

```
public class Ejemplo3 {
    public static void main(String[] args) {
        int i = 45;
        int j=2;
        // Imprime la cadena de caracteres "Cos i : " concatenado con el resultado de calcular el coseno de i
        System.out.println ("Cos i : " + Math.cos(i));
        System.out.println ("Sen i : " + Math.sin(i));
        System.out.println ("j^i : " + Math.pow(j,i));
    }
}
```

Salida obtenida

```
Cos i : 0.5253219888177297
Sen i : 0.8509035245341184
j^i : 3.5184372088832E13
```

1.6. Operadores lógicos

Operador	Operación que realiza		Operador	Operación que realiza
<	Menor que		==	Test de igualdad
>	Mayor que		!=	Test de desigualdad
<=	Menor igual que		&&	And lógico
>=	Mayor igual que		|| (Alt + 124)	Or lógico
!	Not lógico			

```
public class Ejemplo4 {
    public static void main(String[] args) {
        int i = 10;
        int j = 30;
        System.out.println ("i = " + i);
        System.out.println ("j = " + j);
    // La operación encerrada entre paréntesis se resuelve y devuelve un valor booleano (true o false) según el caso
        System.out.println ("i > j es " + (i > j));
        System.out.println ("i < j es " + (i < j));
        System.out.println ("i >= j es " + (i >= j));
        System.out.println ("i <= j es " + (i <= j));
        System.out.println ("i == j es " + (i == j));
        System.out.println ("i != j es " + (i != j));
        System.out.println ("(i < 15) && (j < 15) es "+ ((i < 15) && (j < 15)) );
        System.out.println ("(i < 15) || (j < 15) es "+ ((i < 15) || (j < 15)) );
    }
}
```

Salida obtenida

```
i = 10
j = 30
i > j es false
i < j es true
i >= j es false
i <= j es true
i == j es false
i != j es true
(i < 15) && (j < 15) es false
(i < 15) || (j < 15) es true
```

1.7. CADENAS DE CARACTERES

En Java no hay un tipo predefinido para cadenas de caracteres. En su lugar hay una clase, String, que es la que soporta las distintas operaciones con cadenas de caracteres. La declaración de un String es:

```
String e;              // declarada no inicializada
String e ="";          // cadena vacía inicializada
String e = "Hola";     // declaración y asignación juntas.
```

Operaciones básicas soportadas por la clase String:

Concatenación

La concatenación en Java es muy sencilla: se realiza con el operador +, es decir "sumando" cadenas de caracteres se obtiene la concatenación de estas. Ejemplo:

```
String saludo = "hola";
String nombre = "Pepe";
String saluda_pepe = "";
saluda_pepe = saludo + nombre;  // saluda_pepe toma el valor holaPepe
```

Si se intenta concatenar una cadena con otro tipo de variable, automáticamente esta última se convierte a String, de tal modo que es perfectamente correcto el siguiente ejemplo:

```
String saludo = "hola";
int n = 5;
saludo = saludo + " " + n;       // saludo toma el valor "hola 5"
```

Subcadenas

En la clase String hay un método que permite la extracción de una subcadena de caracteres desde otra.
Su sintaxis es:

Nombre_String.substring((int)posición_inicial, (int)posición_final);

Donde posición_inicial y posición_final son respectivamente la posición del primer carácter que se desea extraer y del primer carácter que ya no se desea extraer.

```
String saludo = "hola";
String subsaludo = "";
Subsaludo = saludo.substring(0,2);       // subsaludo toma el valor "ho"
```

Puede extraerse un char de una cadena. Para ello se emplea el método **charAt**(posición), siendo posición la posición del carácter que se desea extraer.

Comparación de cadenas

Se emplea otro método de String: equals.
Su sintaxis es:

cadena1.equals(cadena2);

Devuelve **true** si son iguales y **false** si son distintos.

El siguiente ejemplo permitirá ilustrar estas operaciones con Strings:

```
public class Ejemplo5 {
   public static void main(String[] args) {
        String saludo = "Hola";
        String saludo2 ="hola";
        int n = 5;

        //Imprime la subcadena formada por los caracteres comprendidos entre el caracter 0 de saludo y hasta el
        //carácter 2, sin incluir el último
        System.out.println (saludo.substring(0,2));

        //Concatena saludo con un espacio en blanco y con el valor de la variable n
        System.out.println (saludo +" " + n);

        //Imprime el resultado del test de igualdad entre saludo y saludo2. Son distintos, en Java se distingue entre
        //mayúsculas y minúsculas.
        System.out.println ("saludo == saludo2 "+ saludo.equals(saludo2));
   }
}
```

Salida obtenida

```
Ho
Hola 5
saludo == saludo2 false
```

1.8. CONVERSIÓN ENTRE STRING y NÚMEROS

Se utilizan en general métodos ofrecidos por las clases ***Wrapper***

1.8.1. Convierte un int a String

```
int unEntero = 3;
String enteroConvertido = String.valueOf(unEntero);
```

1.8.2. Convierte un doble a String

```
double unDoble = 2.25;
String dobleConvertido = String.valueOf(unDoble);
```

1.8.3. Convierte un String a double.

```
String unString = "2.25";
```

1.8.3.1. Forma 1

```
double unDoble = Double.valueOf(unString);
```

1.8.3.2. Forma 2

```
double unDoble = Double.parseDouble(unString);
```

1.8.4. Convierte un String a int.

```
String unString = "2";
```

1.8.4.1. Forma 1

```
int unEntero = Integer.valueOf(unString);
```

1.8.4.2. Forma 2

```
int unEntero = Integer.parseInt (unString);
```

1.8.5. Convierte un String a float.

```
String unString = "3.132";
```

1.8.5.1. Forma 1

```
float unFloat = Float.valueOf(unString);
```

1.8.5.2. Forma 2

```
float unFloat = Float.parseFloat(unString);
```

1.9. ARRAYS

Un array es una colección ordenada de elementos del mismo tipo, que son accesibles a través de un índice.
La sintaxis de declaración de un array es la siguiente:
Tipo_datos[] nombre_array;

Tipo_datos es el tipo de los datos que se almacenarán en el array (int, char, String... o cualquier objeto).

Ejemplos: int[] edades;
Persona[] p;

El segundo ejemplo declara un array que contendrá referencias a objetos de tipo Persona.

En Java los arrays son un objeto, instancias de una clase. Como tales se crean mediante el operador new (la creación se puede dar en la misma línea de la declaración, o en una distinta).

Tipo_datos[] nombre_array = new Tipo_datos[tamaño_array];

Tamaño_array indica la cantidad de elementos que contendrá el array.

Ejemplo: int[] edades = new int[10];

Este ejemplo define un array llamado edades, en el que se podrán almacenar 10 datos de tipo entero.
En el momento de la creación del array se dimensiona el mismo y se reserva la memoria necesaria.
También puede crearse de forma explícita asignando valores a todos los elementos del array en el momento de la declaración, de la siguiente forma:

```
int [ ] edades = { 55 , 31 , 25 };
```

El acceso a los elementos del array se realiza indicando entre corchetes el elemento del array que se desea, teniendo en cuenta que siempre el primer elemento del array es el índice 0.
Por ejemplo:

```
int miEdad = edades[1]
```

Si se intenta usar un índice que está fuera del rango válido para ese array se produce un error de 'Indice fuera de rango'. En el ejemplo anterior se produce esta excepción si el índice es menor que 0 o mayor que 2.
Se puede conocer el número de elementos de un array usando la variable length.
Ejemplo: int cantiElem = edades.length → **cantiElem** contiene el valor 3.

Si se desea realizar un bucle que recorra los elementos de este array el código sería:

```
for(int i= 0; i< 3; i++){
   System.out.println("Elemento " + i + edades[i]);
}
```

Es posible declarar arrays de más de una dimensión. Los conceptos son los mismos que para los arrays monodimensionales.
Por ejemplo:

```
int [ ][ ] a = { { 1 , 2 } , { 3 , 4 } , { 5 , 6 } };
int x = a[1][0];  // x contiene 3
int y = a[2][1];  // y contiene 6
```

Se pueden recorrer los elementos de un array multidimensional, de la siguiente forma:

```
int [ ][ ] a = new int [3][2];
for ( int i = 0 ; i < a.length ; i++ ) {
   for ( int j = 0 ; j < a[i].length ; j++){
      a[i][j] = i * j;
   }
}
```

Obsérvese en el ejemplo la forma de acceder al tamaño de cada dimensión del array.

2. CONTROL DE FLUJO EN JAVA

2.1. SENTENCIAS CONDICIONALES

2.1.1. Sentencia if

Ejecutan un código u otro en función de que se cumpla o no una determinada condición. La condición es una expresión booleana que devuelve true ó false.

```
If(condicion) {
        Grupo de sentencias
}
else{
        Grupo2 de sentencias
}
```

Si condición toma el valor true se ejecuta **Grupo de sentencias**, en caso contrario se ejecuta **Grupo2 de sentencias**. En ambos casos se continúa ejecutando el resto del código.

```
If(condicion) {
        Grupo de sentencias
}
else if (condicion2){
        Grupo2 de sentencias
    }
else if (condicion3){
        Grupo3 de sentencias
    }
.
.
else{
        Grupo_n de sentencias
}
```

Si *condición* toma el valor true se ejecuta **Grupo de sentencias**, sino, si *condicion2* toma el valor true se ejecuta **Grupo2 de sentencias**... y así sucesivamente hasta acabarse todas las condiciones.
Si no se cumple ninguna se ejecuta Grupo_n de sentencias. Este último **else** es opcional. En ambos casos se continúa ejecutando el resto del código.

```
public class Ejemplo6 {
    public static void main(String[] args) {
        // Se imprime el resultado de ejecutar varias veces el método test.
          System.out.println("Con 10 y 5: "+ test(10, 5));
          System.out.println("Con 4 y 9: "+ test(4, 9));
          System.out.println("Con 5 y 5: "+ test(5, 5));
    }
          // Método que podrá ser invocado como test(int a, int b) y que devolverá -1 si a < b, +1 si a > b y 0 si a == b.
          static int test(int val, int val2) {
              int result = 0;
              if(val > val2){
                  result = 1;
               }
              else if(val < val2){
                  result = -1;
              }
              else{
                  result = 0;
              }
              return result;
          }
}
```

Salida obtenida

```
Con 10 y 5: 1
Con 4 y 9:  -1
Con 5 y 5:   0
```

2.1.2.Sentencia Switch

```
switch(selector) {
        case valor_1 :
                Grupo de sentencias_1;
                break;
        case valor_2 :
                Grupo de sentencias_2;
                break;
        ….
        case valor_n:
                Grupo de sentencias_n;
                break;
        default: statement;
}
```

Se compara el valor de *selector* con *valor* del case. Si el valor coincide se ejecuta su respectivo grupo de secuencias. Si no se encuentra ninguna coincidencia se ejecutan las sentencias de default. Si no se colocan los *break*, una vez que se encontrase un valor que coincida con el *selector* se ejecutarían todos los grupos de sentencias, incluída la del default.
Este tipo de estructura tiene posibilidades muy limitadas, ya que en las condiciones sólo admite la igualdad. Además esta comparación de igualdad sólo admite valores tipo char o cualquier tipo de valores enteros menos long.

```
public class Ejemplo7 {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            switch(i) {
                case 0:  System.out.print(i + ": cero");     break;
                case 1:  System.out.print(i + ": uno");      break;
                case 2:  System.out.print(i + ": dos");      break;
                case 3:  System.out.print(i + ": tres");     break;
                default:  System.out.print(i + ": cuatro"); break;
            }
        }
    }
}
```

Salida obtenida

```
0: cero
1: uno
2: dos
3: tres
4: cuatro
```

2.2. BUCLES

Permiten repetir un bloque de código mientras se cumpla una determinada condición.

2.2.1. Bucle while

Cuando en la ejecución de un código se llega a un bucle while se comprueba si se verifica su condición. Si se verifica se continua ejecutando el código del bucle hasta que esta deje de verificarse. Su sintaxis es:

```
while(condición){
        Grupo de sentencias
}
```

```
public class Ejemplo8 {
    public static void main(String[] args) {
        double r = 0;
        //Mientras que r < 0.99 sigue ejecutando el cuerpo del bucle.
        while(r < 0.99) {
                // Genera un nuevo r aleatario entre 0 y 1
                r = Math.random();
                System.out.println(r);
        }
    }
}
```

Salida obtenida

```
0.6526892020767742
0.3280581527892381
0.1704052393117107
0.21053583942917742
0.9942780700232954
```

2.2.2. Bucle do while

Su comportamiento es semejante al bucle while, sólo que aquí la condición va al final del código del bucle, por lo que se garantiza que el código se va a ejecutar al menos una vez. Dependerá del caso concreto si es más conveniente emplear un bucle while o do while. La sintaxis de do while es:

```
do {
        Grupo de sentencias;
}while(condición);
```

2.2.3. Bucle for

```
for(inicialización;expresion; incremento){
        Grupo de sentecias;
}
```

inicialización es una asignación de un valor a una variable: la variable-condición del bucle.
expresion es la condición que se le impone a la variable del bucle para finalizar
incremento indica una operación que se realiza en cada iteración a partir de la primera, sobre la variable del bucle.

```
public class Ejemplo9 {
  public static void main(String[] args) {
        int[] a = { 55 , 31 , 25 };
        for( int i = 0; i < a.length; i++){
            System.out.println("elemento: " + i + " -> " + a[i]);
        }
   }
}
```

Salida obtenida

```
elemento: 0 -> 55
elemento: 1 -> 31
elemento: 2 -> 25
```

3. INGRESO DE DATOS POR TECLADO

3.1. A través del argumento del método main

En la firma del método main:

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

el parámetro del método es lo que se encuentra entre paréntesis. El identificador para el dato de entrada es "`args`" y el tipo de datos es un array de String, es decir, una lista de elementos del mismo tipo, todos del tipo cadena de caracteres.
Los datos que se ingresan por teclado son asignados al array. Deben ingresarse entre comillas dobles (“ “). Para acceder a cualquier elemento de la lista o estructura se utiliza un índice. La numeración para la estructura comienza en 0. El primer elemento es args[0].

Ejemplo: Ingresar por teclado la LU, el nombre y la nota de un alumno

```
{"2157","Juan Perez","7.50"}
```

```
public class IngresoPorMain{
   public static void main(String[] args){
    int LU = Integer.valueOf(args[0]);
```

```
    String nombre = args[1];
    double nota = Double.parseDouble(args[0]);
    System.out.println ("LU: " + LU + "  Nombre: " + nombre + "  Nota: " + nota);
   }
}
```

3.2. Usando la clase Scanner

Esta clase fue implementada a partir del JDK 1.5 (5.0). Se encuentra en el paquete java.util.
Tiene varios constructores, algún tipo de fuente de caracteres. Uno de ellos es la entrada de datos desde consola. A través de sus métodos va seleccionando tokens separados por espacio en blanco. Algunos de estos métodos, una vez leído un token, lo intentan interpretar como algún tipo primitivo de java:

int nextInt()	long nextLong()	short nextShort()	boolean nextBoolean()
double nextDouble()	float nextFloat()		byte nextByte()

Dichos métodos intentan interpretar el token que toca leer, lanzando una excepción si no puede:

- InputMismatchException si el token no responde al tipo deseado
- NoSuchElementException si no quedan más tokens

Ejemplo: Ingresar por teclado dos números de tipo double y mostrar en pantalla su suma.

```
import java.util.Scanner;

public class LeoDoble{
  public static void main(String args []){
      Scanner texto = new Scanner(System.in);
      texto.useDelimiter("\n"); // permite ingresar cadenas separadas por espacio

      System.out.print("Escriba dos números dobles: ");
      double x = texto.nextDouble();
      double y = texto.nextDouble();
      System.out.println("Suma: " + (x + y));
  }
}
```

Cabe mencionar el método

```
String nextLine()
```

que devuelve lo que queda por leer de la línea actual; es decir, desde el lugar adonde esté apuntando, hasta el primer fin de línea. Llamadas consecutivas a nextLine() van proporcionando líneas sucesivas de texto