1 Elementos esenciais da linguaxe Java

1.1 Sumario

1.2 Estrutura dun programa Java

Calquera programa que creemos en Java ten, polo menos, unha clase cun método especial que se chama **main**, tal e como vemos no seguinte exemplo:

```
public class Programa {
      public static void main(String args[ ]) {
            // corpo do programa
      }
}
```

O método main é un método especial que lle di á JVM que clase é a principal dentro do programa. O significado do código anterior é o seguinte:

- public. Indica que o método main() é de acceso público para todo o mundo
- static. Indica ao compilador que main() é o primeiro que debe executarse
- void. Indica que main()non devolve ningún valor
- String args[ ]. Almacena os parámetros que seguen á execución do programa, é dicir, os valores introducidos por **liña de comandos**. Estes parámetros son valores de entrada que lle pasamos ao programa e poden ser números enteiros, reais, caracteres, etc. Sempre se lerán como Strings, é dicir, como cadeas de caracteres. A conversión correspondente (de String a enteiro, por exemplo.) realízase dentro do programa, como veremos.

O seguinte programa recibe dous números por liña de comandos e calcula a súa suma:

```
public class Suma {
 public static void main(String[] args) {
   int x;
   int y;
   int sum;
   x = Integer.parseInt(args[0]); // Conversión a enteiro
   y = Integer.parseInt(args[1]);
   sum = x + y;
   System.out.println("A suma de "+ x +" e "+ y +" é "+ sum);
 }
}
```

Para compilar o programa hai que escribir nunha consola:

```
javac Suma.java
```

Se cando escribimos un programa cometemos erros o compilador avisaranos mediante mensaxes. É importante ler estas mensaxes para detectar e corrixir os problemas e conseguir que o programa funcione.

Para executar o programa, unha vez compilado, hai que chamar ao intérprete tecleando por liña de comandos os números que queremos sumar:

```
java Suma 4 5
```

A saída do programa será:

```
A suma de 4 e 5 é 9
```

A sintaxe completa cando chamamos ao compilador de Java é a seguinte:

```
javac [opcións] NomeFicheiro1.java [NomeFicheiro2.java...]
```

A sintaxe completa cando chamamos á máquina virtual de Java, é dicir, ao intérprete, é a seguinte:

```
java [ opcións ] nomedaclase <argumentos>
```

Ademais do compilador javac e o intérprete java temos a ferramenta **javadoc** que xera documentación en formato HTML a partir do código fonte dun programa. Existen tres formas de introducir comentarios pero só unha delas será a que use javadoc á hora de xerar documentación:

```
// comentario nunha liña
/* comentario nunha ou o máis liñas */
/** comentario de documentación */
```

Os comentarios que javadoc busca no código fonte empezan por **/**** e rematan por ***/**. Existe un conxunto de indicadores que javadoc recoñece como especiais e que empezan por @:

- @author
- @version

A sintaxe de javadoc é a seguinte:

```
javadoc [ opcións ] [ ficheiro.java ]
```

1.3 Variables

Unha variable é unha posición de memoria que almacena un valor. Dende o punto de vista do programador as variables almacenan valores dun determinado tipo (**tipo de datos**) e noméanse cun identificador, tal e como vemos no seguinte exemplo:

```
public class Bicicleta {
      int marcha;
      int velocidade;
}
```

As variables marcha e velocidade son de tipo enteiro. En Java hai catro categorías de variables:

- **Variables de instancia**, tamén chamadas atributos non estáticos. Representan o estado dun obxecto, é dicir, dunha instancia. Tamén se lles chama campos.
- **Variables de clase**, tamén chamadas atributos estáticos. Pertencen a unha clase e non a un obxecto concreto.
- **Variables locais**. Teñen sentido dentro dun bloque de código determinado, un método, un bucle, etc.
- **Parámetros**. Son as variables que se usan para pasar informacion aos métodos.

Os obxectos almacenan o seu estado en variables que chamamos **atributos**.

1.3.1 Tipos de datos

Cada variable ten un tipo que determina os valores que pode tomar e as operacións que se poden realizar sobre ela. Java é unha **linguaxe fortemente tipada** o que quere dicir que as variables e os seus tipos se teñen que declarar antes de usarse, aínda que non teñen que inicializarse.

En Java distinguimos tipos de datos primitivos e tipos de datos referencia:

- Os **tipos primitivos** (todos con signo) poden ser:
  - Enteiros. Hai catro tipos en función do rango de valores que poden tomar:
    - byte. Están codificados con 8 bits. Polo tanto poden represetar valores entre -2^8 e (2^8)-1.
    - short. Están codificados con 16 bits.
    - int. Están codificados con 32 bits.
    - long. Codificados con 64 bits.
  - Reais. Poden declararse como float (precisión simple) ou como double (dobre procesión).
  - Carácter. Decláranase con char.
  - Booleanos. Decláranase coa palabra reservada boolean.

Algúns exemplos de literais de tipos primitivos son os seguintes:

```
178: tipo int
2L: o valor decimal 2 como tipo long
077L: o valor octal 77 como tipo long.
0xBAACL: 0x indica un valor en hexadecimal.
37.266D: Número real de dobre precisión
87.363F: Número real de precisión simple
'c': O carácter c
'\t': O carácter de tabulación
true: O valor booleano verdadeiro
false: O valor booleano falso
```

- Os **tipos referencia**, que poden ser:
  - Arrays
  - Clases
  - Interfaces

1.3.2 Nomes de variables

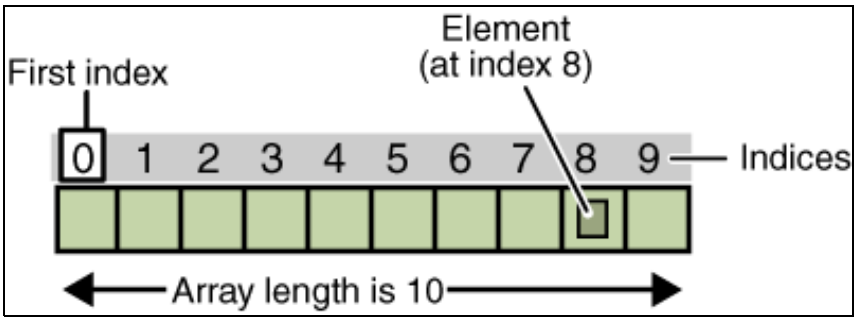
O nome dunha variable ten que ser un identificador válido, é dicir, unha serie ilimitada de caracteres Unicode que empecen por unha letra, o carácter $ ou _. Non poden ser unha palabra clave da linguaxe e non podemos repetir o nome dunha variable no seu **ámbito** (*scope*).

Java é *case-sensitive*, é dicir, distingue entre maiúsculas e minúsculas. Por convención, os nomes de variables empezan cunha letra minúscula e o das clases cunha letra maiúscula. Se o nome dunha variable ten máis dunha palabra xúntanse e cada palabra despois da primeira empeza por maiúscula. Por exemplo:

```
boolean esVisible;
```

1.3.3 Arrays

Un array é un obxecto que contén un número finito de elementos dun mesmo tipo. A lonxitude do array establécese cando se crea e é fixa:



Un array empeza sempre en 0. Para declarar un array hai que seguir as mesmas regras que para a declaración de variables de tipo primitivo, é dicir, declárase o tipo do array e o seu nome, pero non hai por que inicializalo:

```
int [] meuArray;
```

No seguinte exemplo vese como se declara, crea, inicializa e accede a un array:

```
int [] meuArray; // Declaración
```

```
meuArray = new int[10]; // Creación
meuArray[0] = 100; // inicializa o primeiro elemento
System.out.println("Elemento 1: " + meuArray[0]);
System.out.println("Elemento 2: " + meuArray[1]);
System.out.println("Elemento 3 " + meuArray[2]);
System.out.println(meuArray.length);
```

O atributo length do obxecto meyArray devolve a lonxitude do array. Tamén se poden declarar arrays multidimensionais.

1.4 Operadores

Como en calquera outra linguaxe de programación Java ten operadores. Estes son símbolos especiais que executan operacións específicas sobre un, dous ou tres **operandos**, devolvendo un resultado. Hai tres tipos de operadores en función do número de operandos que utilizan:

- 1 operando: operadores unarios
- 2 operandos: operadores binarios
- 3 operandos: operadores ternarios

A existencia de varios operadores nunha expresión fai necesaria determinar a súa **precedencia**, é dicir, en que orde se avalían. Non é o mesmo 3 + 5 * 4, que dará como resultado 23, que (3 + 5) * 4, que será 60.

1.4.1 Operadores de asignación e aritméticos

O operador de asignación permite que unha variable tome un valor concreto. Por exemplo:

```
int i = 0;
```

Os operadores aritméticos permite sumar, restar, multiplicar e dividir, usando os símblos **+**, **-**, *****, /, respectivamente. Os operadores aritméticos pódense combinar co operador de asignación para obter asignacións compostas. Por exemplo:

```
x+=1 e x=x+1 incrementan o valor de x en 1
```

O operador + tamén se pode usar para concatenar dúas cadeas, tal e como vemos no seguinte exemplo:

```
class ConcatDemo {
    public static void main(String[] args){
         String primeiraCadea = "Isto é";
         String segundaCadea = " unha cadea concatenada.";
         String terceiraCadea = primeiraCadea + segundaCadea;
         System.out.println(terceiraCadea);
    }
}
```

O operador aritmético **%**, tamén chamado módulo, devolve o resto enteiro dunha división. Por exemplo: 11%5 devolve 1

1.4.2 Operadores unarios

Requiren un único operador e son os seguintes:

- Operador de incremento **++**: incrementa un valor en 1, por exemplo: i++
- Operador de decremento **--**: decrementa un valor en 1, por exemplo: i--
- Operador de complemento lóxico **!**: inverte o valor dun boolean, é dicir, se é true convérteo en false e viceversa

Nos operadores de incremento non é o mesmo facer i++ (**operador postfixo**) que ++i (**operador prefixo**), tal e como vemos no seguinte exemplo:

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

```
  }
}
```

1.4.3 Operadores de igualdade e relacionais

Úsanse para determinar se un operando é maior, menor, igual ou distinto a outro operando. Son os seguintes:

- **==** Igual a
- **!=** Non igual a
- **>** Maior que
- **>=** Maior ou igual que
- **<** Menor que
- **<=** Menor ou igual que

Un erro frecuente cando se empeza a programar en Java é usar o operador de asignación **=** en lugar de **==**, e viceversa.

1.4.4 Operadores condicionais

Avalían dúas ou máis expresións devolvendo como resultado verdadeiro ou falso. No xaso de varios operandos (expresión) só se avalía o segundo operando en caso necesario. A isto chámaselle curtocircuíto. Os operadores condicionais son os seguintes:

- **&&** AND condicional
- || OR condicional

Por exemplo:

```
int x = 1;
int y = 2;
if((x == 1) && (y == 2)) System.out.println...
if((x == 1) || (y == 1)) System.out.println...
```

1.4.5 Operador instanceof

Permite comprobar se un obxecto é unha instancia dunha clase, dunha subclase ou dunha clase que implementa un determinado interface. Por exemplo:

```
class InstanceofDemo {
 public static void main(String[] args) {
   Pai obx1 = new Pai();
   Pai obx2 = new Fillo();
   System.out.println("O obx1 é unha instancia da clase Pai()? " + (obx1 instanceof Pai));
   System.out.println("O obx1 é unha instancia da clase Fillo()? " + (obx1 instanceof Fillo));
   System.out.println("O obx1 é unha instancia de MeuInterface? " + (obx1 instanceof MeuInterface));
   System.out.println("O obx2 é unha instancia da clase Pai()? " + (obx2 instanceof Pai));
   System.out.println("O obx2 é unha instancia da clase Fillo()? " + (obx2 instanceof Fillo));
   System.out.println("O obx2 é unha instancia de MeuInterface? " + (obx2 instanceof MeuInterface));
 }
}

class Pai{} // Esta clase non ten métodos nin atributos

class Fillo extends Pai implements MeuInterface{}

interface MeuInterface{}
```

1.4.6 Operadores de manexo e desprazamento de bits

Transforman os operadores a binario e realizan as operacións traballando cos bits un a un. Son os seguintes:

- **<<**. Desprazamento á esquerda. Exemplo: A << B Desprazamento de A á esquerda en B posicións
- **>>**. Desprazamento á dereita. Exemplo: A >> B Desprazamento de A á dereita en B posicións
- **&**. AND lóxico. Exemplo: A&B. Operación AND bit a bit
- |. OR lóxico. Exemplo: A|B. Operación OR bit a bit
- **^**. OR exclusivo ou XOR. Exemplo: A^B. Operación XOR bit a bit

- ~. Complemento a 1. Exemplo: ~A. Invirte os bits de A

1.5 Expresións, sentenzas e bloques

Xa vimos que os **operadores** traballan con **operandos**. Os operadores úsanse para construír **expresións**. Por exemplo, 2 + 3 é unha expresión que unha vez avaliada toma o valor 5. As expresións son o elemento fundamental das **sentenzas** que, á súa vez, se agrupan en **bloques**.

Unha expresión pode ser unha simple variable, ou operandos con operadores ou unha chamada a un método. Constrúense segundo a sintaxe da linguaxe. Cada expresión é avaliada polo compilador e devolve un valor. O tipo de dato devolto depende do elemento usado na expresión. Algúns exemplos de expresións son os seguintes:

```
int marcha = 0;
unArray[0] = 100;
System.out.println("Elemento 1: " + unArray[0]);
int result = 1 + 2;
if(valor1 == valor2)
System.out.println("value1 == value2");
```

Cando usamos expresións compostas hai que ter en conta a precedencia dos operadores. Non é o mesmo x+y*100 que (x+y)*100. Se queremos que x se sume a y antes de que sexa multiplicado por 100 teremos que usar parénteses.

Unha sentenza representa unha unidade de execución e remata sempre cun **;**. As sentenzas nas linguaxes de programación son coma as oracións gramaticais nas linguaxes naturais. En Java hai tres tipos de sentenzas:

- **Sentenzas de control de fluxo**: Permiten alterar a execución normal dun programa
- **Sentenzas de declaración**: Declaran unha variable, i.e: int i = 0;
- **Sentenzas de expresións**: As sentenzas de expresión proveñen, como o seu nome indica, dunha expresión e poden ser de catro tipos: expresións de asignación; calquera uso de ++ ou --; invocación de métodos ou expresións de creación de obxectos. Veamos un exemplo:

```
unValor = 8933.234;  // Sentenza de asignación
unValor++;               // Sentenza de incremento
System.out.println("Olal, meu!");     // Sentenza de invocación a un método
Bicicleta unhaBicicleta = new Bicicleta(); // Sentenza de creación dun obxecto
```

Un bloque é un conxunto de cero ou máis sentenzas e sempre comeza por unha chave aberta, carácter **{**, e remata cunha chave pechada, carácter **}**. Nun programa Java un bloque pode usarse en calquera lugar onde estea permitido usar unha sentenza simple. Por exemplo:

```
class BlockDemo {
    public static void main(String[] args) {
         boolean condicion = true;
         if (condicion) { // Comeza o bloque I
              System.out.println("Condición é true.");
         } // Fin do bloque I
         else { // Comeza o bloque II
              System.out.println("Condición é false.");
         } // Fin do bloque II
    }
}
```

1.6 Control de fluxo

A orde normal de execución dun programa é secuencial, é dicir, as sentenzas que conforman o programa vanse executando dunha en unha secuencialmente. Existen, non entanto, situacións nas que é necesario alterar esta orde de execucución, para elo utilizamos as sentenzas de control de fluxo que permiten alterar o fluxo normal de execución dun programa en función dunha condición.

Neste tipo de sentenzas avalíase unha expresión e en caso de que o resultado sexa true realízase unha determinada acción. Hai varias sentenzas deste tipo.

1.6.1 Sentenza if-then (se-entón)

Existe unha **cláusula if** onde se avalía unha expresión. En función do valor que tome esa expresión (true ou false) execútase a **cláusula then** (un bloque de sentenzas) ou sáltase á sentenza seguinte. Vexamos un exemplo: No seguinte método

```
void usaFreos(){
```

```
    if (estaMovendose){ // Cláusula if
         velocidade--; // Cláusula then
    }
}
```

Se despois da cláusula if hai unha única sentenza non é necesario poñer as chaves de comezo e fin de bloque.

1.6.2 Sentenza if-then-else (se-entón-se non)

Funciona igual que a sentenza if-then pero proporciona un segundo camiño de execución se a cláusula if-then é avaliada como false. No seguinte método, usaFreos(), se a expresión estaMovendose é true a variable velocidade decrementarase. Se non, imprímese por pantalla unha mensaxe indicando que a bicicleta xa está parada.

```
 void usaFreos(){
       if (estaMovendose){ // Cláusula if
          velocidade--; // Cláusula then
       }
       else { // '''Cláusula else'''
          System.err.println("A bici xa está parada!");
       }
     }
   }
```

1.6.3 Sentenza switch

Se temos moitas posibles opcións para escoller a sentenza if-then pode ser confusa. Existe outra sentenza de conrol de fluxo, chamada switch, que permite escoller entre un número ilimitado de opcións. Switch traballa cos tipos de datos primitivos byte, short, char e int. Ao corpo dunha sentenza switch chámaselle un **bloque switch**. Unha sentenza **break** sae do bloque switch. Por exemplo:

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

        int mes = 8;
        switch (mes) {
            case 1:  System.out.println("Xaneiro"); break;
            case 2:  System.out.println("Febreiro"); break;
            case 3:  System.out.println("Marzo"); break;
            case 4:  System.out.println("Abril"); break;
            case 5:  System.out.println("Maio"); break;
            case 6:  System.out.println("Xuño"); break;
            case 7:  System.out.println("Xullo"); break;
            case 8:  System.out.println("Agosto"); break;
            case 9:  System.out.println("Setembro"); break;
            case 10: System.out.println("Outubro"); break;
            case 11: System.out.println("Novembro"); break;
            case 12: System.out.println("Decembro"); break;
            default: System.out.println("Mes non válido.");break;
        }
    }
}
```

1.6.4 A sentenza while

A sentenza while executa un bloque de sentenzas mentres unha expresión (condicición) sexa avaliada a true. Polo tanto é unha **sentenza repetitiva**. Sintacticamente exprésase así:

```
while (expresión) {
  sentenza(s)
}
```

Se a condición nunca cambia teremos un **bucle infinito**. Salvo excepcións isto non é o que buscamos polo que haberá que modificar a expresión condición dentro do bloque while para que nalgún momento o programa poda saír do bucle. Por exemplo:

```
class WhileDemo {
     public static void main(String[] args){
          int contador = 1;
          while (contador < 11) {
               System.out.println("O contador vale: " + contador);
```

```
                contador++;
            }
        }
    }
```

Podemos implementar un bucle infinito co seguinte código:

```
while (true) {
  // Código a executar
}
```

1.6.5 Sentenza do-while

É outra sentenza repetitiva que tamén permite facer bucles. Sintacticamente exprésase así:

```
do {
  sentenza(s)
} while (expresión);
```

A principal diferenza con while e que este bucle **sempre se executa unha vez**.

1.6.6 Sentenza for

A sentenza for tamén permite facer bucles. Proporciona unha forma compacta de iterar sobre un rango de valores. Sintacticamente exprésase así:

```
for (inicialización; condición; incremento) {
  sentenza (s)
}
```

A expresión que inicializa o bucle execútase sempre unha vez, cando empeza o bucle. Cando a expresión se avalía a false o bucle remata. A expresión de incremento invócase en cada iteración do bucle e pódese aumentar o valor ou decrementalo. Un exemplo de uso do bucle for é o seguinte:

```
class ForDemo {
     public static void main(String[] args){
          for(int i=1; i<11; i++){
               System.out.println("O contador vale: " + i);
          }
     }
}
```

1.6.7 Sentenza break

A súa función é saír dun bloque, por exemplo, dentro do switch visto anteriormente, pero, ademais, a sentenza break remata os bucles for, while e do-while. Por exemplo:

```
class BreakDemo {
    public static void main(String[] args) {
        int[] arrayDeInts = { 32, 87, 3, 589, 12, 1076,
                              2000, 8, 622, 127 };
        int numeroBuscado = 12;
        int i;
        boolean atopado = false;

        for (i = 0; i < arrayDeInts.length; i++) {
            if (arrayDeInts[i] == numeroBuscado) {
                atopado = true;
                break;
            }
        }

        if (atopado) {
            System.out.println("Atopado " + numeroBuscado
                               + " no índice " + i);
        } else {
            System.out.println(numeroBuscado
                               + " non está no array");
        }
    }
```

}