Introdución a Java

Índice

1.- Introdución a Linguaxe Java

Java é unha linguaxe puramente orientada a obxecto, de herdanza simple, de raíz única e pseudocompilado.

- **Puramente orientada a obxecto**: En Java unicamente existen obxectos.... salvo uns elementos existentes por motivos de optimización e velocidade denominados ***tipos primitivos.*** Todos os tipos primitivos dispoñen de clases equivalentes.
- **Herdanza simple**: En Java unha clase soamente pode herdar dunha clase pai. Por exemplo, en Java non é posible definir a clase ***ProfesorUniversitario*** que sexa ao mesmo tempo un ***Profesor*** e un ***Investigador*** a pesar de que debe ter todos os atributos e métodos dun Profesor e todos os de Investigador. Outras linguaxes orientadas a obxectos como C++ si o permiten.
- **Raíz Única**: Todos os obxectos Java pertencen a clase *Object*. Todas as clases que se definen en Java herdan da clase Object de xeito automático, e polo tanto teñen todos os seus atributos e métodos.
- **Pseudocompilado**: O código fonte se compila a un código máquina propio da máquina virtual java (*JVM*)

1.1 – A Máquina virtual de Java JVM, ficheiros .jar e o ClassPath

A máquina virtual Java (JVM) é unha implantación mediante software dunha CPU capaz de executar un código máquina propio denominado *p-code*. O código fonte que escribe o programador se debe compilar ao código máquina que corresponde con esta CPU antes de ser executado.

> En Java debemos gardar cada ***clase pública*** nun ficheiro que se chame igual que a clase. Cada clase definida se corresponde con un ficheiro en código máquina (.class) sin importar si están definidas ou non no mesmo ficheiro.

> A máquina virtual java se inicia mediante o comando ***java*** que leva como parámetro o código máquina que queremos executar (o arquivo .class).
>
> O compilador java se invoca mediante o comando ***javac,*** que leva como parámetro o ficheiro fonte que desexamos compilar. Si ese código fonte fai uso de máis ficheiros se compilarán tamén de xeito automático.

Cando creamos unha aplicación Java non programamos todo o código necesario, se non que habitualmente facemos uso de clases xa creadas presentes la ***libraría estándar de Java*** ou por terceiros que nos ofrecen funcionalidade necesaria para a nosa aplicación. Cando executamos a nosa aplicación mediante o comando ***java aplicación*** se debe cargar na JVM todo o p-code necesario (o que programamos nos e o das librarías) a partir dos ficheiros *.class*.

> A JVM buscará eses ficheiros .class nas rutas indicadas na variable do contorno **CLASSPATH**. Tamén é posible indicar o classpath mediante o parámetro ***-cp <directorios separados por comas>*** cando executamos a aplicación.

Distribuír e instalar unha aplicación composta de multitude de ficheiros .class residentes en diferentes carpetas é bastante problemático e propenso a erros. Unha mellor opción é crear un ficheiro comprimido que conteña as carpetas e os ficheiros .class da aplicación xunto co resto de recursos que precise (imaxes, sons... etc). Java utiliza para esta función o formato ***.jar***

> Un arquivo ***.jar*** e un conxunto de carpetas comprimidas que conten unha serie de recursos Java: clases, imaxes... etc.

1.2 – As anotacións @

As anotacións representan instrucións para o compilador. O compilador java (javac) interpretará as liñas que comezan por @ para optimizar a compilación ou realizar operacións alternativas. Este tipo de comandos se coñecen como ***instrucións do preprocesador*** que é a parte do compilador que se encarga de interpretalas antes de proceder a compilación, facendo posible cousas como a inxección de código na compilación facilitando a creación de utilidades como *JUnit* ou *Hibernate*. Unha anotación típica é ***@Override*** que lle indica ao compilador que o método é unha versión sobreposta da clase base o que permite verificar posibles erros antes de compilar.

> A ***inxección de código*** na compilación fai referencia a que o compilador inserta código fonte "prefabricado" antes de compilar substituíndo a anotación.

Outro exemplo son as anotacións de Javadoc como @param o @return que permiten xerar documentación a partir dos comentarios mediante unha utilidade denominada ***javadoc***.

1.3 – O primeiro programa: "Ola Mundo"

Java é unha linguaxe puramente orientada a obxectos, polo tanto todo o que podemos facer é definir clases. Todo o código ten que pertencer a unha clase. Polo tanto o programa mínimo en Java sería algo así:

```
public class  AMiñaPrimeiraClase  {
}
```

Este programa debería residir nun ficheiro denominado *AMiñaPrimeiraClase.java* e se pode compilar a código máquina co comando ***javac AMiñaPrimeiraClase.java*** xerando o código máquina *AmiñaPrimeiraClase.class.* Pero NON é posible executar o código xa que non ten "punto de entrada", o sistema non sabe por onde debe comezar a executar o código. Si queremos poder executar o programa, o mínimo sería algo así:

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

Esta versión, a diferencia da anterior, si dispón de "punto de entrada". Polo tanto, podemos executar o programa mediante o comando ***java AMiñaPrimeiraClase*** despois de compilalo con ***javac AMiñaPrimeiraClase.java*** que produce o arquivo en código máquina ***AMiñaPrimeiraClase.class***

> O método ***public static void main(String[] arg)*** é o punto de inicio do algoritmo que executa a máquina virtual java cando invocamos a unha clase. A JVM buscará na clase ese método e levará a cabo o algoritmo codificado.

Si probamos o programa anterior veremos que non fai nada. A aplicación ten punto de comezo, pero non temos un algoritmo polo que o programa chega ao final e remata a execución. Si queremos algún efecto deberíamos "mandar" facer algo como por exemplo visualizar "Ola Mundo!".

```
public class  AMiñaPrimeiraClase  {
   public static void main(String[] args) {
            System.out.println("Ola Mundo!");
   }
}
```

2.- Os espazos de Nomes: Os Packages

> Os espazos de nomes pretenden evitar colisións entre os identificadores utilizados para nomear as clases.

En Java todas as clases que definimos pertencen a un espazo de nomes. Si non indicamos nada ese espacio de nomes é o espazo "***default***" ou package por defecto. Non é boa idea crear clases no espazo de nomes por defecto, porque nos expoñemos a posibles colisións a pouco que desenvolvamos unha cantidade significativa de código. Deberíamos crear as clases nun espazo de nomes significativo que indique de algún modo:

- A quen pertence o código
- A que "área" pertence o código (contabilidade, gráficos, xogos, utilidades .... etc)

O nome elixido para o espazo de nomes debería ser ***único no mundo*** xa que non sabemos quen poderá facer uso das nosas clases a medio ou longo prazo.

> As URI expresadas de forma inversa son unha boa idea a hora de seleccionar nomes para o package, xa que garanten ser únicas no mundo, indican a quen pertence o código (ao usar un nome de dominio) e permiten indicar fácilmente o área de aplicación da clase.

En Java, o espazo de nomes se indica cunha liña ao principio do código indicando ***package nome.do.package;***. Ademáis de evitar a colisión de nomes, o package nos indica como debe estar organizado o código.

> Cada .class debe atoparse na carpeta indicada no package. Deste xeito, unha clase pertencente ao package *com.iesrodeira.development.games* debería estar na carpeta *com/iesrodeira/development/games* contando dende a carpeta de inicio da aplicación.

Cando dende unha clase queremos facer uso de outra que está en un package diferente temos dúas posibilidades:

a) Utilizar o nome completo da clase antepoñendo o nome do package.
b) Importando a clase mediante **import** ao principio do código e despois da definición do package. Import soporta comodíns como * que permiten importar múltiples clases cunha soa liña.

> Os packages non son xerárquicos, incluír as clases do package ***com.iesrodeira.datos.**** non inclúe as clases do package ***com.iesrodeira.datos.poboación***

3.- Identificadores, Constantes e Variables: Expresións aritméticas e lóxicas

3.1 - Identificadores

> Un identificador é un nome que identifica un obxecto ou da nome a unha clase

En Java a creación de identificadores debe seguir unha serie de normas:

- Un identificador debe ser único no seu ámbito
- Un identificador pode ter a lonxitude que se queira
- Un identificador consiste en letras maiúsculas e minúsculas e díxitos do 0 ao 9. Se consideran unha "letra" os caracteres do 'A' a 'Z' de 'a' a 'z', o signo $ e o guión baixo _
- Un identificador debe comezar por unha letra
- Por norma xeral non se usan nin $ nin _

Existe unha convención para a creación de identificadores en Java que é moi aconsellable seguir:

1. Si un identificador consta de varias palabras, cada unha das palabras comeza por maiúscula salvo a primeira letra.
2. Os nomes das clases e *enums* comezan por maiúscula.
3. Os nomes dos atributos e variables comezan por minúscula
4. As constantes (atributos estáticos finais) se escriben en maiúscula.
5. Os identificadores deben ser significativos e breves.

3.2 - Variables

> Unha variable é un nome (identificador) que se lle da a un espazo de memoria onde podemos gardar ou recuperar información. Java é unha linguaxe de ***tipos fortes***, polo tanto, antes de utilizar unha variable é necesario definila indicando o seu tipo.

Un ***tipo de datos*** indica o espazo de memoria utilizado para gardar a información e a forma en que se deben interpretar os bits que este espazo contén. Para definir unha variable se escribe o tipo de datos que é capaz de almacenar seguido do identificador que se utilizará para acceder a información almacenada:

```
int x;
double y;
```

3.3 - Constantes

As constantes non identifican espazos de memoria, se non que identifican datos concretos. Na maioría das linguaxes o compilador substitúe o identificador polo dato antes de traducir o código fonte a código obxecto. No caso de Java, non existen realmente as constantes como tal. Pero podemos definir variables indicando que unha vez que lles asignamos un valor na definición ou na creación do obxecto, ese valor non poderá variar. Para iso se utiliza a palabra reservada ***final.***

```
final double PI = 3.141592;
```

3.4 Expresións Aritméticas e Lóxicas

Unha expresión é unha combinación de valores ou variables (que conteñen valores) e operadores que producen un resultado ou provocan unha acción. As expresións aritméticas son as que producen resultados numéricos mentres que as expresións lóxicas son as que unicamente poden chegar ao resultado verdadeiro (true) ou falso (false).

> Os operadores aritméticos e lóxicos teñen unha ***precedencia*** que define a orde na que se deben avaliar as operacións indicadas mediante os diferentes operadores. Para alterar esa precedencia podemos facer uso do paréntese ().

- Os operadores aritméticos permiten facer operacións matemáticas como a *, /, %, + , - ...
- Os operadores lóxicos permiten facer operacións lóxicas como >=, <=, ==, != ...
- O operador de asignación asigna o resultado dunha expresión a unha variable =
- Podemos incluír a chamada a un método ou función como parte dunha expresión.

Os operadores dispoñibles ordenados por precedencia son os seguintes:

Descrición	Operadores	Asociatividad
Paréntese	***( )***	De esquerda a dereita
Creación de instancias	***new***	De esquerda a dereita
Acceso a atributo	***Identificador do obxecto.identificador do atributo***	De esquerda a dereita

Descrición	Operadores	Asociatividad
Acceso a táboa	***Identificador da táboa[]***	De esquerda a dereita
Invocación de método	***Identificador do obxecto.identificador do método(parámetro, parámetro...)***	De esquerda a dereita
Post Incremento	***variable++, variable--***	-
Pre Incremento	***++variable, --variable***	-
Operadores Unarios	***+expr, -expr, ~expr, !expr***	De dereita a esquerda
Multiplicación, división, resto	****, / , %***	De esquerda a dereita
Suma, resta	***+, -***	De esquerda a dereita
Movemento de Bits	<<, >>, >>>	De esquerda a dereita
Relacionais	>, <, <=, >=, ***instanceof***	De esquerda a dereita
Igualdade, desigualdade	== , ***!***=	De esquerda a dereita
AND a nivel de bit	***&***	De esquerda a dereita
XOR a nivel de bit	^	De esquerda a dereita
OR a nivel de bit	|	De esquerda a dereita
AND lóxico	***&&***	De esquerda a dereita
OR lóxico	||	De esquerda a dereita
Condicional abreviado	***? :***	De esquerda a dereita
Asignación	= , *******=, /=, ***%***=, +=, -=, <<=, >>=, >>>=, ***&***=, ^=, |=	De esquerda a dereita

4.- Os tipos de datos: Tipos Primitivos e Obxectos

4.1 – Os tipos primitivos

Java é unha linguaxe puramente orientada a obxectos, o que quere dicir que ***todos os elementos dos programas son obxectos que pertencen a unha clase***. Sen embargo, por motivos de optimización e velocidade existen uns poucos tipos de datos que non son obxectos se non variables "normais" nas que se almacena un valor. Estes tipos primitivos son:

byte: Valor numérico de 8 bits
char: character en formato UTF-8 de 16 bits
short: Número enteiro de 16 bits
int: Número enteiro de 32 bits
long: Número enteiro de 64 bits
float: Número real de 32 bits en formato de coma flotante IEEE 754
double: Número real de 64 bits en formato de coma flotante IEEE 754
boolean: Valor booleano true(1) ou false(0) ocupando 1 bit.

A información destes tipos se pode representar tamén mediante obxectos das clase *Byte, Boolean, Short, Integer, Long, Float, Double e Boolean* respectivamente. E posible converter entre os tipos primitivos e obxectos da clase correspondente de xeito automático e transparente nun proceso coñecido como "***autoboxing***" mentres que converter entre un obxecto dunha destas clases e o tipo primitivo de xeito transparente se chama "***unboxing***".

4.1.1 – Conversión de tipos

O tipo da variable determina a información que se pode almacenar na mesma. Non é posible almacenar un valor de un tipo nunha variable definida como de outro tipo salvo que o formato da información almacenada sexa compatible (números enteiros, coma flotante...). Si a conversión non supón perda de información (de un tipo que precisa menos bytes a un tipo que precisa máis) se pode facer a asignación sen problemas mentres que a inversa se necesita facer un "casting" que indique explicitamente que queremos realizar a conversión:

```
int x=23;
long y=54;
y=x;  //  NON precisa casting
x=(int) y; // SI precisa casting
```

4.2 – Os Obxectos

Un obxecto é un elemento membro dunha clase. E dicir, contén todos os atributos e métodos indicados na definición da clase. Podemos pensar na clase como o "tipo" da variable que vai a almacenar a referencia do obxecto.

Por "***referencia***" queremos dicir a posición da memoria RAM onde se sitúa realmente a información

En Java todos os obxectos se crean de xeito dinámico. Cando definimos unha variable como un obxecto, o que creamos realmente é unha variable capaz de almacenar a ***referencia*** de un obxecto de esa clase.

Por exemplo, en Java: **Rectangle r;**

Non crea ningún obxecto (en outras linguaxes como C++ sí se crearía o obxecto, que estaría almacenado na variable ***r***). O que crea esta definición é un espazo onde se pode almacenar a referencia dun obxecto da clase *Rectangle*, pero o obxecto aínda non existe.
Para crear o obxecto, empregamos o operador ***new*** para realizar un proceso coñecido como instanciación do obxecto:

r=new Rectangle(); *// Crea un obxecto da clase Rectangle e almacena a posición de memoria onde reside en r*

Rectangle() fai referencia a invocación dun método "especial" que teñen todas as clases denominado ***construtor***. A misión do construtor é a creación dun obxecto da clase. Si non definimos os nosos propios construtores, se xera un construtor sen argumentos denominado ***construtor por defecto***.

4.2.1 – Os Arrays

Un "***Array***" ou táboa é un conxunto de datos do mesmo tipo aos que podemos acceder mediante un índice.

En Java os arrays son obxectos, polo tanto debemos crealos empregando o operador ***new*** :

```
int[] x;                        // Define un array de "int" de 1 dimensión ou vector
Rectangle[] tiles;              // Define un array de "Rectangle" de 1 dimensión ou vector
Rectangle[][] mosaic;           // Define un array de "Rectangle" de 2 dimensións ou matriz
int[][][] coord3D;              // Define un array de "int" de 3 dimensións
```

/* As definicións anteriores unicamente crean variables capaces de almacenar referencias a arrays dos tipos indicados, pero non crean ningún array. Debemos crealos mediante ***new*** */

x=new int[10]; // Crea un array de 10 elementos de tipo int e almacena a súa referencia en x
tiles=new Rectangle[8]; // Crea un array de 8 elementos de tipo Rectangle e almacena a súa referencia en tiles
mosaic=new Rectangle[4][10]; // Crea un array de 4 elementos no que cada un de eles é un array de 10 elementos de tipo Rectangle e almacena a súa referencia en mosaic
coord3D=new int[15][20][30]; // Crea un array de 15 elementos no que cada un de eles é un array de 20 elementos de tipo array de 30 elementos de tipo int e almacena a súa referencia en coord3D

/* As sentencias anteriores crean os arrays, pero non os obxectos (no caso de Rectangle). Para crear os obxectos, debemos empregar new de novo */

tiles[5]=new Rectangle(3,4); // Crea un obxecto Rectangle de lados de lonxitude 3 e 4 e almacena a súa referencia na posición 5 do array tiles
Rectangle r=tiles[2]; // Almacena en r a referencia do Rectangle almacenada na posición 2 do array tiles

int o=coord3D[3][5][10]; // Almacena en o o valor int almacenado na posición 10 do array que está na posición 5 do array que está na posición 3 do array coord3D
Coord3D[6][0][9]=134; // Almacena o número int 134 na posición 9 do array que está na posición 0 do array que está na posición 6 do array coord3D

Para almacenar ou recuperar información nos arrays utilizamos ***índices***. O índice indica a posición do array a que queremos acceder.

O número de índices necesario para acceder a un elemento do array se denomina ***dimensión do array***

Como podemos ver nas definicións anteriores, traballar con varias dimensións pode ser bastante "difícil" de explicar. Resulta muy útil representar as táboas de 1 dimensión como unha fila de celdas, as de dúas dimensións como unha retícula composta de filas e columnas, e as de 3 como un cubo composto de filas, columnas e profundidade. As táboas de máis de 3 dimensións son de raro uso, e son difíciles de representar.

5.- As sentencias de control

En Java se poden agrupar instrucións en "bloques" mediante o emprego de { }. As chaves definen un ***ámbito*** que serve para agrupar instrucións en bloques como para definir a vida das variables. ***As variables definidas nun bloque unicamente teñen existencia no propio bloque.***

5.1 – Selección

En Java podemos distinguir dúas estruturas equivalentes de selección (é posible utilizar unicamente a estrutura if..else) e unha estrutura abreviada.

5.1.1 – IF e IF abreviado

Permite executar un grupo de instrucións si unha expresión lóxica é certa, ou outro si é falsa.

```
if (expresión lóxica) {
        Bloque de instrucións
}
```

```
if (expresión lóxica) {
        Bloque de instrucións
} else {
        Bloque de instrucións
}
```

Tamén existe a "***versión abreviada***": ***(expresión lóxica)?instrución/resultado si se cumple:instrución/resultado si non se cumple***

```
/* Si x > 0 entonces suma 1 a y, se non resta 1 a y */
if (x>0) {
    y=y+1;
} else {
    y=y-1;
}
```

```
/* Si x > 0 entonces suma 1 a y, se non resta 1 a y */
y=(x>0)?y+1:y-1;
```

> **"else"** únicamente é necesario especificalo si queremos facer algo no caso en que a expresión condicional se avalíe a ***false***

5.1.2 – switch

A sentencia ***switch*** nos permite levar a cabo un grupo de instrucións dependendo do valor dunha variable:

```
switch(variable) {
        case valor:
                instrucións
                break;
        case valor:
                instrucións
                break;
        .....
        default:
                instrucións
                break;
}
```

```
....
String nome;
switch(dia) {
        case 1:     nome="Luns";
                    break;
        case 2:     nome="Martes";
                    break;
        case 3:     nome="Mércores";
                    break;
        ....
}
```

> O ***switch*** debe empregarse en situacións nas que é necesario realizar distintas accións dependendo do valor que tome unha variable. Un caso típico é realizar a acción elixida nun menú de opcións.

5.2 - Iteración

En Java existen tres estruturas de iteración, aínda que é posible utilizar a estrutura ***while*** en todos os casos:

```
while(condicion) {
        bloque de instrucións
}
```

```
do {
        bloque de instrucións
} while(condicion);
```

```
for(precondicion;condicion;postcondicion) {
        bloque de instrucións
}
```

6.- A entrada e saída na consola

Para comprender o funcionamento da entrada e saída por consola en Java é necesario coñecer o concepto de ***Stream*** ou fluxo de datos.

> Un ***Stream*** e un fluxo de datos que vai dende un orixe a un destino. Un ***Stream de entrada*** é un fluxo de datos do que podemos *ler* información.
>
> Un ***Stream de saída*** é un fluxo de datos no que podemos *escribir* información.

6.1 – A clase System

A clase ***System*** pertence a libraría de clases Java e ten métodos e atributos estáticos útiles para realizar operacións estándar no sistema. En particular ten tres atributos estáticos que son Streams de datos asociados aos dispositivos de entrada e saída do sistema:

- **err**: E un ***Stream de Saída*** (no que podemos escribir) da clase PrintStream normalmente asociado coa pantalla do sistema e utilizado para **visualizar erros.**
- **out**: E un ***Stream de Saída*** (no que podemos escribir) da clase PrintStream normalmente asociado coa pantalla do sistema e utilizado para **visualizar información.**
- **in**: E un ***Stream de Entrada*** (do que podemos ler) da clase InputStream normalmente asociado ao teclado e utilizado habitualmente para **solicitar información** ao usuario.

A clase ***PrintStream*** nos proporciona métodos prácticos que podemos utilizar directamente, deste xeito, podemos visualizar información coa seguinte sentencia: ***System.out.println("Hola Mundo!");***
A clase ***InputStream*** a diferencia de PrintStream non ten métodos que útiles directamente para a entrada de datos por teclado, polo que se normalmente se utilizan clases como BufferedReader ou o máis práctica Scanner.
A clase Scanner proporciona distintos métodos útiles para a entrada de datos por teclado, sendo o mais útil ***String nextLine()*** que nos retornará o String introducido por teclado.

> Creación dun obxecto ***Scanner*** que sirva para ler datos do teclado: ***Scanner scn=new Scanner(System.in);***

> **Scanner** proporciona métodos como ***int nextInt()*** ou ***float nextFloat()*** pero é moito máis práctico ler a información con ***String nextLine()*** para posteriormente converter os datos con int ***Integer.parseInt(String str)*** ou ***Float Float.parseFloat(String str)***

7.- Definición de Clases

Unha clase é unha definición dun tipo de obxectos indicando as características ou atributos propios dos obxectos da clase e as súas "capacidades" ou métodos. Os atributos se indican mediante variables mentres que os métodos se codifican mediante funcións. Para definir unha clase se emprega a palabra reservada ***class***

```
class MyFirstClass {
}
```

Unha vez definida a clase, podemos crear obxectos que posúen todas as características e capacidades definidas en *MyFirstClass* invocando ao construtor mediante ***new***: ***MyFirstClass fobject=new MyFirstClass()***

Esta sentencia crea un obxecto de tipo *MyFirstClass* na memoria RAM e almacena a súa posición de memoria na variable ***fobject***. Ese obxecto terá todas as características definidas en *MyFirstClas*s e todas as súas capacidades.

7.1 - Definición de Atributos

As características dos obxectos se definen mediante variables. Por exemplo podemos establecer que os obxectos pertencentes a clase *MyFirstClass* deben ter unha descrición de tipo String...:

```
class MyFirstClass {
        String description;
}
```

7.1.1 – Atributos estáticos

Cando definimos unha clase, cada obxecto terá valores propios nos seus atributos. Por exemplo, un obxecto da clase MyFirstClass pode ter no seu atributo *description* o String "*Obxecto 1*", e outro o valor "*Obxecto 3*".

> O valor de todos os atributos dun obxecto nun instante concreto se coñece como ***estado do obxecto***

Sin embargo é posible crear atributos que en lugar de pertencer a cada obxecto creado pertenzan a clase. Deste xeito, o atributo ten existencia e pode ter un valor sen necesidade de crear ningún obxecto. Son atributos globais a todos os obxectos da clase. Este tipo de atributos se denominan "***estáticos***" e se marcan coa palabra reservada ***static.***

> Se debe intentar evitar o uso de atributos estáticos salvo causas moi xustificadas, xa que o seu abuso produce código propenso a erros e difícil de manter.

7.2 - Definición de Métodos.

Os métodos describen a funcionalidade ofrecida polos obxectos dunha clase. Os métodos poden recibir información en variables locais denominadas *parámetros*, e producir un resultado ou *valor de retorno.*
A definición dun método en Java se realiza do seguinte xeito:

```
tipo_do_valor_de_retorno  identificador_do_método(lista_de_parámetros) {
        /* Algoritmo do método codificado en Java */
}
```

Si o método non produce ningún valor de retorno se indica coa palabra reservada ***void***, e si non precisa parámetros se deixa vacío o contido entre paréntese. Para retornar un valor se utiliza a palabra reservada ***return.***

return valor; provoca a finalización do método e o retorno do valor especificado.

7.2.1 – Métodos estáticos: main

Do mesmo xeito que podemos declarar atributos de clase ou estáticos é posible definir métodos de clase mediante a palabra reservada ***static***

Os métodos static pertencen a clase, e non precisan dun obxecto para ser utilizados. Poden chamarse antepoñendo o nome da clase seguido do método.

Para acceder os atributos e métodos dun obxecto se utiliza o operador punto (.), que separa a variable que referencia o obxecto do atributo ou método ao que queremos acceder. Por exemplo para acceder ao atributo *description* dun obxecto ao que fai referencia unha variable ***fobject*** escribiremos *fobject.description,* e para invocar ao seu método *showDescription* escribiremos *fobject.showDescription()*

7.3 – Creando Obxectos. O operador new

Unha vez definidas as clases dos obxectos que precisa a nosa aplicación, podemos crealos mediante o operador ***new***. Este operador reservará unha zona da memoria RAM onde se almacenará principalmente o estado do obxecto (o valor dos seus atributos) e retornará a posición ou referencia de esa zona de memoria que podemos almacenar nunha variable.

Definición da clase Rectangle:

```
class  Rectangle {
    private double lenhorizontal;
    private double lenvertical;

    public Rectangle(double lenhorizontal,double lenvertical) {
        this.lenhorizontal=lenhorizontal;
        this.lenvertical=lenvertical;
    }

    public double getArea() {
        return lenhorizontal*lenvertical;
    }

    public double getPerimetro() {
        return lenhorizontal*2+lenvertical*2;
    }
}
```

Creación dun obxecto Rectangle:

Rectangle r=new Rectangle(123.12,45.00);

A sentencia anterior crea unha instancia da clase Rectangle (obxecto) nunha zona da memoria RAM e almacena a dirección de memoria desa zona na variable **r**.

A definición da variable ***Rectangle r*** non crea ningún obxecto, simplemente reserva na memoria RAM un espazo de memoria capaz de almacenar a dirección de memoria dunha instancia da clase Rectangle. Mentres non poñamos unha referencia válida non podemos saber o que está na dirección almacenada en ***r*** (un espazo da memoria, e decir un conxunto de bytes, *sempre ten un valor*)

7.3.1 - Os construtores e destrutores: A recolección de lixo

Cando creamos un obxecto co operador **new** debemos indicar a continuación o construtor que queremos utilizar para inicializar o estado do obxecto.

Un **construtor** é un método que se chama igual que a clase sen especificar un tipo de datos de retorno, e que se encarga de crear o obxecto na memoria e de poñer valores iniciais aos atributos. Os atributos para os que non se especifique un valor inicial tomarán o valor 0.

Moitas linguaxes orientadas a obxectos tamén teñen un método **destrutor** que se executa sempre que se destrúe un obxecto. Java sen embargo non ten ***destrutores*** xa que o instante no que se destrúe un obxecto non pode preverse de modo fiable de ningún xeito. Java ten un sistema de *recolección de lixo* que a JVM utiliza cando precisa liberar memoria. Este sistema se encarga de eliminar os obxectos que xa non teñen utilidade para a aplicación (xa que ningunha variable fai referencia a eles e polo tanto son inaccesibles).

Si cando deseñamos unha clase non creamos un método construtor, a JVM proporciona un sen argumentos denominado construtor por defecto. Si polo contrario creamos un construtor, a JVM non facilitará ningún construtor.

7.3.2 – As referencias a obxectos: O problema da comparación

Como xa comentamos, cando as variables en Java non almacenan obxectos, se non que almacenan a referencia da posición de memoria onde se atopa o obxecto. En cambio as variables de tipos primitivos non almacenan referencias, se non que almacenan o valor desexado. Isto ten varios efectos que poden pasar desapercibidos:

Este código visualizará os valores 2 e 5:

```
int x=3;
int y=2;

x=y;
y=5;
System.out.println(x);
System.out.println(y);
```

Este código visualizará os valores 5 e 5:

```
class Number {
   private int value;

   public Number(int v) {
            this.value=v;
   }

   public void setValue(int v) {
            this.value=v;
   }

   public int getValue() {
            return this.value;
   }
}
```

```
Number x=new Number(3)
Number y=new Number(2);

x=y;
y.setNumber(5);
System.out.println(x.getValue());
System.out.println(y.getValue());
```

Esta expresión lóxica valerá *true*:

```
int x=3;
int y=3;

if (x == y) System.out.println("Iguais!!");
```

Esta expresión lóxica será *false*:

```
Number x=new Number(3)
Number y=new Number(3);

if (x == y) System.out.println("Iguais!!");
```

Isto se debe a que as variables non almacenan o obxecto, se non a referencia de memoria onde está o obxecto. Cando copiamos o contido dunha variable en outra, o que facemos e que as dúas variables apunten ao mesmo obxecto xa que o que copiamos é a referencia.
Cando comparamos dúas variables, tampouco estamos a comparar os atributos do obxecto, se non que o que estamos a facer e comparar as referencias. En lugar de comprobar si dous obxectos "son iguais" o que estamos comprobando é si "son o mesmo obxecto".
Para solucionar este problema a clase Object (da que herdan todas as clases Java) dispón dun método denominado **equals** que retorna ***true*** cando o obxecto recibido no parámetro se considera igual que o obxecto actual e ***false*** en outro caso: ***public boolean equals(Object obj);***

Resulta evidente que a consideración de que dous obxectos "son iguais" e subxectiva. ¿ Cando consideramos iguais dous coches ? ¿ Cando coincide marca e modelo ? ¿ Debe coincidir tamén a cor ?....

> As clases deben sobrepoñer o método ***equals*** para realizar a comparación entre os obxectos da clase do xeito correcto.

Por outra banda a copia de obxectos non se pode realizar co operador de asignación xa que iso so copiaría a referencia. Si queremos obter unha copia dun obxecto o mellor modo de facelo e creando un ***construtor de copia.***

> Un **construtor de copia** é un construtor que recibe como argumento un obxecto da clase e que copia nos atributos do obxecto que se está a crear os valores dos atributos do obxecto recibido.

Un exemplo claro do potencial problema das referencias é o seguinte:

```
class Exemplo {
  private int testA;
  private int[] testB;

   public Exemplo() {
      testA=10;
      // Se podería facer testB=new int[] {10};
      testB=new int[1];
      testB[0]=10;
   }

   public int getTestA() {   return testA;   }
   public int[] getTestB() {    return testB;   }

   @Override
   public String toString() {
      return "VALOR testA: "+testA+"  VALOR testB: "+testB[0];
   }
}
```

```
public static void main(String[] args) {
      Exemplo e=new Exemplo();
      int x=e.getTestA();
      x=55;  // Isto non altera o contido do atributo testA do obxecto e
      int[] y=e.getTestB();
      y[0]=55; // Isto SI altera o contido do atributo testB do obxecto e
      System.out.println(e);
   }
}
```

7.4 – Os Enum

Un enum non é un tipo especial de clase da que non é posible crear novos elementos salvo os indicados na súa definición. Polo tanto, ***os elementos do enum son únicos e constantes***. Os seguintes exemplos amosan un enum básico e un enum un pouco máis elaborado.

```
enum DiasSemana { LUNS, MARTES, MÉRCORES,XOVES,VENRES,SÁBADO,DOMINGO };
```

```
enum Meses {
 Xaneiro(31),Febreiro(28),Marzo(31),Abril(30),Maio(31),Xuño(30),Xullo(31),Agosto(31),Setembro(30),Outubro(31),Novembro(30),Decembro(31);

 public int dias;

  Meses(int ndias) {
            this.dias=ndias;
 }
};
```

Como os elementos do enum son únicos e constantes se poden comparar mediante o operador ==

8.- Creando unha Aplicación Java

Podemos realizar unha aplicación Java empregando un procesador de texto plano ou un IDE (Integrated Development Environment). Si desenvolvemos a aplicación cun procesador de texto plano, debemos manter a estrutura de directorios marcada polos ***packages*** que definamos. Unha vez creado o código fonte, o podemos compilar co comando ***javac ClasePrincipal.java*** e executalo co comando ***java ClasePrincipal*** sempre que as variables de contorno CLASSPATH estea correctamente configurada., noutro caso podemos pasar as carpetas das clases java mediante **java -cp lista de carpetas ClasePrincipal.**

O uso de un IDE proporciona numerosas vantaxes respecto ao uso de editores de texto planos:

- **Xestión de Proxectos**: As aplicacións constitúen ***proxectos*** que son empaquetados en arquivos ***.jar*** de fácil distribución.
- **Integración de sistemas de control de versións** como CVS, Subversion ou Git
- **Refactorización de Código**, permitindo renomear identificadores en todos os ficheiros do proxecto.
- Automatización de Código, permitindo a inserción automática de código estándar como construtores, setters, getters....
- **Detección temprana de erros de sintaxe**: O Compilador avisará dos posibles erros de sintaxe a medida que escribimos o código
- **Facilidade de compilación e execución**: Cun simple click podemos compilar e executar a aplicación sen abandonar o contorno de desenvolvemento.
- **Axuda en Liña**: O IDE informa dos métodos e atributos das clases que empregamos.

8.1 - O método main e os seus parámetros

Cando lanzamos a máquina virtual Java (JVM) debemos indicar a clase que queremos que a JVM execute. A JVM buscará nesa clase un método estático coa firma ***public static void main(String[] args)*** e levará a cabo o algoritmo programado nese método. O argumento ***String[]*** contén os parámetros que se subministren cando se lance a JVM. Por exemplo, si lanzamos a JVM indicando a clase MainApplication.class do seguimte xeito:

java MainApplication parametro1 parametro2 parametro3

A ***JVM*** buscará dentro da clase *MainApplication* o método *public static void main(String[] args)* e lle pasará en args o array cos argumentos { parametro1, parametro2, parametro3 } de xeito que args[0] terá o valor parametro1, args[1] o valor parametro2 e args[2] o valor parametro3. A lonxitude do array args será 3.

9.- A Herdanza: A clase Object

9.1 – A clase Object

Mediante herdanza podemos deseñar clases que herdan as características e métodos da clase pai. Java é unha linguaxe orientada a obxectos pura e de raíz única. Isto quere dicir que toda a informaciónque xestiona un programa Java son obxectos, e que todos os obxectos descenden de unha única clase base: ***Object***

9.1.1 – toString, equals e getClass

Algúns dos métodos máis útiles que as clases herdan de Object son ***getClass***, ***toString*** e ***hashCode*** e ***equals***.

- **Class<?> getClass();** retorna o obxecto Class que representa a clase á que pertence o obxecto. Os obxectos da clase Class. Tamén é posible acceder a ese obxecto class mediante ***NomedaClase.class***

> Mediante o método ***static Class<?> Class.forName(String classname)*** podemos recuperar o obxecto Class que corresponde a unha clase polo seu nome forzando así a carga na JVM

- **public String toString();** se encarga de retornar a representación como String do obxeco. A representación programada na clase Object retorna *nomedaclase@direcciónndememoria* polo que debemos sobrepoñer o método si queremos un comportamento distinto.
- ***public int hashCode();*** se encarga de retornar un número único para cada obxecto que se considera distinto. E importante para axilizar a comparación con equals e para o almacenamento en estruturas complexas como ***HashMap***.
- **public boolean equals(Object object);** se encarga de determinar si o obxecto recibido como parámetro se considera "igual" que o obxecto actual. A implantación en Object se limita a retornar ***true*** so si son o mesmo obxecto, polo que é necesario sobrepoñelo nas clases derivadas que precisen un comportamento distinto.

> Os IDE son capaces de xerar de xeito automático *construtores, setters, getters, hashCode, equals e toString.*

```
public class Cliente {
    private String nif;
    private String nomeapelidos;
    private String dirección;
    private String email;
    private String telefono;

    public Cliente(String nif, String nomeapelidos, String dirección, String email, String telefono) {
        this.nif = nif;
        this.nomeapelidos = nomeapelidos;
        this.dirección = dirección;
        this.email = email;
        this.telefono = telefono;
    }

    @Override
    public int hashCode() {
        int hash = 7;
        hash = 71 * hash + Objects.hashCode(this.nif);
        return hash;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Cliente other = (Cliente) obj;
        if (!Objects.equals(this.nif, other.nif)) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return nomeapelidos+" ("+nif+")";
    }
}
```

```
public class TestCliente {
    public static void main(String[] args) {
        Cliente c1=new Cliente("0000000","Cliente 1","suKasa","su@email","sutelefono");
        Cliente c2=new Cliente("0000001","Cliente 1","suKasa","su@email","sutelefono");
        Cliente c3=new Cliente("0000000","Un cliente diferente a 1","otraKasa","otro@email","otrotelefono");

        System.out.println(c1+" y "+c2+" son: "+((c1.equals(c2))?"iguais":"distintos"));
        System.out.println(c1+" y "+c3+" son: "+((c1.equals(c3))?"iguais":"distintos"));

    }
}
```

```
Cliente 1 (0000000) y Cliente 1 (0000001) son: distintos
Cliente 1 (0000000) y Un cliente diferente a 1 (0000000) son: iguais
```

9.2 – Definindo clases herdadas: extends

Para crear unha clase derivada a partir dunha clase base en Java se utiliza a palabra reservada ***extends.***

```
public class Animal {
   protected String name;

   public void talk() {
      System.out.println("......");
   }

   public void eat() {
      System.out.println(name+" is eating");
   }
}
```

Creamos a clase herdada Dog
E un programa que crea un Dog
e chama os seus métodos talk e eat

```
public class Dog extends Animal {
}
```

```
public class Fauna {
   public static void main(String[] args) {
      Animal animal=new Animal();
      Dog dog=new Dog();

      animal.talk();
      animal.eat();
      dog.talk();
      dog.eat();
   }
}
```

```
......
null is eating
......
null is eating
```

9.2.1 – super e o construtor por defecto

Como vemos no exemplo anterior, non temos ningún construtor en Animal nin en Dog, polo tanto se utiliza un construtor por defecto sin argumentos. Si creamos un construtor en Animal o sistema deixará de subministrar un construtor por defecto, e a clase Dog deixará de funcionar xa que non pode crear a clase base:

```
public class Animal {
   protected String name;

   public Animal(String name) {
      this.name=name;
   }

   public void talk() {
      System.out.println("......");
   }

   public void eat() {
      System.out.println(name+" is eating");
   }
}
```

necesitamos poñer un construtor en Dog que chame cree correctamente o Animal que é Dog

```
public class Dog extends Animal {

   public Dog() {
      super("Dog");
   }

   /** Podemos facer tamén isto
    *  Sobrecargando o constructor
    */
   public Dog(String name) {
      super(name);
   }
}
```

```
public class Fauna {
   public static void main(String[] args) {
      Animal animal=new Animal("beast");
      Dog dog=new Dog();

      animal.talk();
      animal.eat();
      dog.talk();
      dog.eat();
   }
}
```

```
......
beast is eating
......
Dog is eating
```

Si en Animal crearamos un construtor sen argumentos, non sería necesario modificar Dog

9.3 – Superposición e Polimorfismo

E posible que o xeito de falar dos Animais Dog sexa distinto que a de outros animais. Podemos ***superpoñer*** o método ***talk*** para que os Animais Dog falen correctamente.
Ademais, como os Dog son Animais podemos facer referencia a eles con variables de tipo Animal en lugar de Dog. O polimorfismo garante que o seu comportamento é correcto.

```
public class Dog extends Animal {
   public Dog() {
      super("Dog");
   }

   public Dog(String name) {
      super(name);
   }

   @Override
   public void talk() {
      System.out.println("Guau, Guau, Guau ....");
   }
}
```

```
public class Fauna {
   public static void main(String[] args) {
      Animal animal1=new Animal("beast");
      Animal animal2=new Dog();

      animal1.talk();
      animal1.eat();
      animal2.talk();
      animal2.eat();
   }
}
```

```
......
beast is eating
Guau, Guau, Guau ..
Dog is eating
```

9.4 – Introspección: A clase Class

Mediante a introspección un obxecto pode obter información sobre a súa propia estrutura. Mediante o método *getClass()* de Object podemos obter o obxecto ***Class*** ao que pertence o obxecto. O API da clase Class proporciona métodos útiles para facer unha introspección detallada si é necesario

> Existe un obxecto Class por cada clase da aplicación chamado ***NomeClase.class***

9.5 – Encapsulación e Protección de acceso

En Java podemos protexer o acceso a atributos e métodos dunha clase coa seguintes palabras reservadas:

- **public** – O método ou atributo é accesible dende todas partes
- **friendly (default)** – E a protección por defecto si non especificamos nada. O método e atributo é accesible dende calqueira clase do mesmo *package*
- **protected** – O método ou atributo é accesible dende a propia clase ou clases herdadas
- **private** – O método ou atributo so é accesible dende a propia clase

> Como xa comentamos, como os obxectos se xestionan con referencias debemos ter coidado cando retornamos unha referencia que queremos privada mediante un *setter*, xa que damos a posibilidade de que se modifique o seu valor de xeito incontrolado.

10.- Control de Erros: Valores de retorno e Excepcións

O xeito máis efectivo de controlar os posibles erros nunha aplicación é xeralmente mediante o uso de excepcións en lugar de valores de retorno. En Java podemos distinguir dous grupos de excepcións:

- **checked Exceptions**: Herdan da clase Exception. *As checked Exception deben ser capturadas ou relanzadas de xeito explícito e obrigatorio*.
- **Unchecked Exceptions:** Herdan da clase Error ou RuntimeException. Non precisan ser capturadas nin relanzadas.

> Cando unha Exception non se captura se relanza ao método anterior ata chegar ao método ***main*** da aplicación. Si ahí non se captura, o programa rompe visualizando na pantalla información sobre o erro e o lugar onde se lanzou.

En Java, para tratar as exceptions se utiliza a estrutura **try {}.** A sentencia try agrupa un bloque de código ao que podemos especificar un ou varios ***catch(Exception e) {}*** e un ***finally {}***. Si no bloque de código se produce unha excepción, a execución saltará ao primeiro bloque ***catch(Exception e) {}*** no que a clase da Exception producida coincida co tipo de Exception especificada no parámetro.

> Si se indica un bloque ***finally {}*** se levará a cabo xusto antes de rematar a execución do método de xeito independente a que se produza un erro ou non. E útil para liberar recursos utilizados no método sin importar que un erro provoque a finalización prematura do mesmo

```
import java.util.Scanner;

public class Fauna {
   public static void main(String[] args) {
      Scanner scn=new Scanner(System.in);
      String[] nums=new String[10];
      int pos;

      nums[3]="Felicidades, atopaches o tesouro";
      try {
         System.out.print("Escribe un número: ");
         pos=Integer.parseInt(scn.nextLine());
         if (pos >= 0) {
            System.out.println(nums[pos]+" ("+nums[pos].length()+")");
         }
      } catch(ArrayIndexOutOfBoundsException ex) {
         System.out.println("O número debe estar entre 1 e 10");
      } catch(NullPointerException ex) {
         System.out.println("Fallaches, ahí non está o texto");
      } catch(NumberFormatException ex) {
         System.out.println("Debes escribir un número enteiro");
      } finally {
         System.out.println("Bye Bye");
      }
   }
}
```

```
/**
* Si non introducimos un número enteiro a saída será:
*     Debes escribir un número enteiro
*     Bye Bye
* Si introducimos un número negativo a saída será:
*     Bye Bye
* Si introducimos un número maior ou igual que 10 a saída será:
*     O número debe estar entre 1 e 10
*     Bye Bye
* Si introducimos un número distinto de 3 a saída será:
*     Fallaches, ahí non está o texto
*     Bye Bye
* Si introducimos o número 3 a saída será:
*     Felicidades, atopaches o tesouro (32)
*     Bye Bye
*/
```

11.- Clases Abstractas e Interfaces

11.1.- Clases Abstractas

Unha **clase abstracta** é unha clase da que non se poden crear obxectos, e que normalmente ten algún método sen programar (método abstracto). Serven de base para herdar de elas e crear clases máis concretas.

Moitas veces sabemos que atributos e métodos debe ter unha clase, pero é tan xenérica que non podemos programar algúns algoritmos de forma xenérica (xa que existen moitas subclases nas que esa acción se leva a cabo de modos distintos). Nese caso debemos crear unha clase abstracta.

```
abstract class Figura {
        private Coordinate[] coords;

        public Figura(int size) {
                coords=new Coordinate[size];
        }

        abstract void draw(Canvas canvas);
}
```

En esta clase non se pode crear o algoritmo paint que se debe encargar de pintar a figura nun obxecto de tipo Canvas porque é demasiado xenérica. Non se pinta igual un rectángulo que unha circunferencia ou un texto.

11.2.- Interfaces

Unha ***interface*** é unha definición dun conxunto de funcionalidades a ser implantadas, unha API. *Unha clase Java so pode herdar dunha clase base, pero pode implantar todas as interfaces que se desexe*.

Moitas veces sabemos que funcionalidades debe ter un conxunto de obxectos, pero non temos ningunha característica propia común a eles, so funcionalidades. Ademais non existe modo de codificar ningunha desas funcionalidades de modo xenérico. Nese caso debemos crear unha interface.

Nas últimas versións de Java se permiten implantacións por defecto dos métodos dunha interface antepoñendo ao método a palabra reservada ***default.***

```
public interface Drawable {
        public void draw(Canvas c);
}
```

*Esta interface indica que un obxecto para poder considerarse Drawable debe dispoñer de un método **public void draw(Canvas c);***

Si temos definido a interface anterior, a clase Figura anterior poderíamos definila do seguinte xeito:

```
abstract class Figura implements Drawable {
        private Coordinate[] coords;

        public Figura(int size) {
                coords=new Coordinate[size];
        }

        @Override
        abstract void draw(Canvas canvas);
}
```

Isto permitiría tratar os obxectos da clase Figura como obxectos da clase Drawable, e empregar un obxecto Figura en calquera sitio que se necesite un obxecto Drawable (que teña un método draw)

11.2.1 - Interfaces funcionais, Clases anónimas, Expresións Lambda e Method References

As interfaces poden implantarse deseñando unha clase e indicando na cabeceira ***implements.*** Pero tamén pode facerse mediante a creación dunha ***clase anónima***. As clases anónimas se definen creando un obxecto ao mesmo tempo que definimos os métodos da clase, e poden crearse partindo de interfaces ou de clases abstractas.

```
Drawable d=new Drawable() {
        @Override
        public void draw(Canvas canvas) {
                System.out.println("Fake Drawable!!!");
        }
}
```

Creamos un obxecto Drawable mediante unha clase anónima e gardamos a súa referencia na variable d

Unha ***interface funcional*** é unha interface que define un único método

Si a interface unicamente ten un método, e sabemos de que clase se trata podemos utilizar unha expresión lambda, omitindo tanto o nome da clase como o nome do método: `Drawable d=(Canvas canvas) -> { System.out.println("Fake Drawable!!!"); };`

Si o método a implantar é compatible con algún método estático existente, pode utilizarse un *Method Reference:*

```
interface Visible {
   public void show(String str);
}
```

Definición da interface Visible

```
Visible printer=new Visible() {
        @Override
        public void show(String str) {
                System.out.println(str);
        }
}
```

Implantación mediante clase Anónima

```
Visible printer=(String str)->{ System.out.println(str); };
```

Implantación mediante expresión Lambda

```
Visible printer=System.out::println;
```

Implantación mediante Method Reference

Os IDE poden transformar de xeito automático entre clases anónimas, expresións lambda e Method References

11.2.2 – Interfaces da libraría Java: Comparable, Runnable, Closeable, Iterable...

Java ten múltiples interfaces na libraría estándar, as máis importantes son as seguintes:

- ***Comparable***: define o método ***int compareTo(T o)***. Os obxectos que implantan *Comparable* poden ser comparados con este método
- ***Runnable***: define método ***void run();***. Os obxectos que implantan *Runnable* poden ser executados (se pode invocar ao seu método run). Este interface se utiliza na creación de Threads (multiproceso)
- ***Closeable***: define o método ***void close();*** Os obxectos que implantan *Closeable* teñen un método close, polo que se poden pechar.
- ***AutoCloseable***: e idéntica a Closeable, pero é fundamental para o uso de ***try with resources***, na que se especifican recursos que deben ser pechados automaticamente.
- ***Iterable:*** define varios métodos, pero todos teñen unha definición por defecto salvo ***Iterator<T> iterator()*** que retorna un obxecto Iterator que permite percorrer os elementos almacenados no obxecto. Este interface permite o uso de ***for*** para percorrer coleccións de datos.

11.3.- Tipos Xenéricos (Generics)

Si nos fixamos na definición da interface Comparable da librería Java veremos isto:

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

que antes da existencia de Generics, sería algo así

```
interface Comparable {
        int compareTo(Object o);
}
```

Sin Generics, para implantar a interface teríamos que empregar ***casting*** para poder facer as comparacións correctamente e construír o algoritmo que realice a comparación. Calquera erro cos tipos de datos non sería detectado ata que se executara o programa. Todos os obxectos son Object pero cando facemos o ***casting*** si non é do tipo apropiado o programa remata cun erro. Para solucionar isto, se poden utilizar tipos de datos xenéricos (como T en este caso) que antes de compilar se substitúen polo tipo correcto:

Sen Generics

```
class Ventas implements Comparable {
        int quantity;

        public void setQuantity(int q) {
                quantity=q;
        }

        @Override
        public int compareTo(Object o) {
                Ventas ov=(Ventas) o;
                return quantity-ov.quantity;
        }
}
```

Con Generics

```
class Ventas implements Comparable<Ventas> {
        int quantity;

        public void setQuantity(int q) {
                quantity=q;
        }

        @Override
        public int compareTo(Ventas v) {
                return quantity-v.quantity;
        }
}
```

Os IDE poden xerar os métodos necesarios para completar unha clase abstracta ou un interface con un simple click do rato.

12.- Estruturas de datos Dinámicas

Para almacenar información na memoria do ordenador dispoñemos ata o momento de dúas posibilidades:

1. Uso de variables simples. Nunha variable se pode almacenar un dato dun tipo concreto. Ese tipo pode ser unha clase (nese caso o dato sería un obxecto), ou un dato de tipo primitivo (int, char, ...).
2. Uso de táboas (arrays). As táboas nos permiten almacenar un conxunto de datos do mesmo tipo, e logo acceder a cada un de eles mediante un índice.

Non obstante, nas aplicacións aparecen necesidades de almacenamento máis complexas cando temos que almacenar un conxunto de datos:

- Espacio de almacenamento dinámico – Eliminando a necesidade de decidir o espazo máximo cando creamos a variable, se non que o espazo dispoñible se adapta ao contido de xeito automático co único límite da memoria RAM
- Facilidade de inserción, busca e borrado – A estrutura de almacenamento proporciona métodos que facilitan enormemente a busca, inserción e borrado de elementos.

Polo tanto, existe a necesidade de estruturas de almacenamento avanzadas que nos proporcionen almacenamento dinámico e funcionalidades extra que faciliten o seu uso. En Java dispoñemos de dúas interfaces que definen estas funcionalidades con varias clases que as implantan: ***Collection*** *e* ***Map***.

12.1 – Collection

Unha ***Collection*** é un grupo de obxectos coñecidos como os seus "elementos". Algunhas "collections" permiten datos duplicados, outras non. Algunhas manteñen os datos ordeados, outras non.

A interface ***Collection*** herda da interface ***Iterable***, que como xa comentamos antes define tres métodos dos que dous xa teñen unha implantación por defecto, e o terceiro Iterator<T> iterator(); debe retornar un obxecto que implante o interface ***Iterator***, que define tres métodos, dos que é imprescindible implantar dous:

- boolean hasNext(); : retorna true si o obxecto ten elementos sen percorrer.
- E next(); retorna o seguinte elemento na iteración

Os obxectos que implantan Iterable son Iterables, e polo tanto se poden percorrer cunha versión específica do bucle for:

```
for (Clase variable: obxecto) {
/* variable vai tomando cada un dos valores almacenados en obxecto */
}
```

Os Array son *Iterable*, e polo tanto podemos facer o seguinte:

```
Cliente[] cl=new Cliente[20];
for (Cliente c: cl) {
        System.out.println("Cliente: "+cl);
}
```

A interface ***Collection*** se divide en dous tipos diferenciados ou subinterfaces ***List*** e ***Set***. A diferencia principal entre eles é que mentres que os obxectos que implantan ***List*** poden repetirse (ser iguais), nos obxectos que implantan ***Set*** os obxectos almacenados non poden repetirse, deben ser únicos.

Entendemos que dous obxectos son iguais si o seu método equals retorna true cando recibe o outro obxecto como parámetro.

12.1.1 – List

A interface ***List*** permite almacenar, recuperar, eliminar e percorrer un conxunto de datos que almacena. E o máis similar a un Array de unha dimensión, pero dotado de almacenamento dinámico e facilidade de inserción e borrado. As principais implantacións de List na libraría estándar Java son: ***ArrayList*** e ***Vector.***

A diferencia entre ***ArrayList*** e ***Vector*** é que Vector permite o acceso concorrente de varios fíos mentres que ArrayList non. A causa de isto, ArrayList é máis eficiente en aplicacións cun so fío.

12.1.2 – Set

A interface ***Set*** é similar a List, pero non admite elementos duplicados. Entre as clases da libraría estándar Java que o implantan destacan ***HashSet*** e ***TreeSet***.

A diferencia entre ***HashSet*** e ***TreeSet*** radica en que ***TreeSet*** manten os datos ordeados ***segundo a orde natural*** dos obxectos almacenados, ou proporcionando ao construtor un obxecto que implante *Comparator* establecendo o criterio de comparación.

> A orde natural está definida para os tipos primitivos, para os obxectos se debe implantar a interface ***Comparable*** definindo o método *compareTo*

12.2 - Map

Un ***Map*** "mapea" chaves a valores, de xeito que mediante a chave podemos recuperar rapidamente o valor. Un ***Map*** non pode ter chaves duplicadas e unha chave "mapea" un único valor.

> Esta interface proporciona os métodos ***Set<K> keySet();*** que retorna un ***Set*** que conten todas as chaves do ***Map***, e ***Collection<V> values();*** que retorna unha *Collection* con todos os elementos gardados no *Map*.

As clases máis importantes da libraría estándar de Java que implantan Map son ***HashMap*** e ***TreeMap***

HashMap almacena valores identificados por unha chave que nos permiten recuperar rápidamente un elemento coñecendo a súa chave. ***TreeMap*** mantén os elementos ordeados por chave segundo a *orde natural* da clase que pertence a chave ou mediante un obxecto Comparator subministrado ao construtor.

13.- Almacenando información en ficheiros

Un sistema informático xestiona a información almacenada na súa memoria RAM en formato binario, sendo procesada pola CPU para producir información nova que é almacenada de volta na memoria RAM do sistema. A memoria RAM é volátil, polo que si desconectamos o equipo a información almacenada se perde. Polo tanto é necesario almacenar a información binaria en algún tipo de soporte de información permanente.

Se entende por soporte calquera dispositivo que almacena información. Poden ser soportes volátiles (como a RAM) ou permanentes (como un SSD ou unha cinta magnética). Os soportes de almacenamento permanente máis utilizados son os soportes magnéticos, que nos últimos anos están sendo substituídos polas memorias SSD.

En canto ao xeito en que permiten almacenar e recuperar a información, podemos clasificar os soportes de almacenamento en:

- ***Soportes Secuenciais:*** A información se almacena unha detrás de outra segundo vai chegando, non é posible retroceder nin avanzar a unha posición de escritura concreta sin pasar por enriba de información previa. Entre estes tipos de soporte podemos destacar as cintas magnéticas.
- ***Soportes de Acceso Aleatorio:*** Permiten acceder a posicións concretas do soporte de xeito directo para realizar operacións de lectura e escritura. De este tipo de soportes podemos destacar os Discos Magnéticos (HDD) ou as memorias de estado sólido (SDD).

Os dispositivos máis comúns de almacenamento utilizados polos programas son soportes de acceso aleatorio, principalmente HDD e SDD. Dende o punto de vista da programación o concepto máis importante é o de bloque físico.

> Un ***bloque físico*** é a cantidade de información mínima que se lee ou escribe no soporte de información. Por exemplo, si utilizamos un bloque físico de 4K isto quere dicir que a información se intercambia entre o disco e a RAM en "anacos" de 4K

Existen varias técnicas para conseguir conservar os datos dun programa en Java. Basicamente a persistencia consiste no almacenamento da información do estado dos obxectos (valores dos atributos) nun dispositivo de almacenamento permanente. O sistema de almacenamento permanente pode ser de calquera tipo: unha BBDD relacional, unha BBDD orientada a obxectos, unha BBDD noSQL, un simple ficheiro secuencial nunha cinta ou nun soporte de almacenamento directo como un disco magnético ou SSD.... etc.

Principalmente podemos adoptar dous enfoques:

- ***Serialización***: Se transforma o obxecto completo nunha secuencia de bytes. Esa secuencia de bytes se pode almacenar nun sistema de almacenamento e permite reconstruír o obxecto no seu estado orixinal.
- ***Almacenamento dos atributos un a un:*** Almacenamos os valores dos atributos necesarios para restaurar o estado do obxecto.

O proceso de converter un obxecto nunha secuencia de bytes se coñece como ***serialización***.

O proceso de converter unha secuencia de bytes nun obxecto no seu estado orixinal se denomina ***deserialización***

A ***persistencia*** consiste no emprego de técnicas para preservar a información o apagado do sistema ou entre distintas execucións do software.

A ***serialización*** dun obxecto consiste na transformación do obxecto a un conxunto de bytes coa información necesaria para reconstruír o obxecto.

Para que un obxecto Java poda converterse automaticamente a un conxunto de bytes debe implantar a interface ***Serializable***.

A interface ***Serializable*** dispón de implementacións por defecto dos seus métodos, de xeito que si queremos que un obxecto sexa serializable bastará con indicar na súa definición que "implements Serializable". Si necesitamos realizar algunha acción "especial" a hora de serializar obxectos dunha determinada clase, podemos sobrepoñer os métodos da interface, aínda que non é habitualmente necesario:

- ***private void writeObject(java.io.ObjectOutputStream out) throws IOException***
- ***private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;***
- ***private void readObjectNoData() throws ObjectStreamException;***

A maioría das clases estándar da libraría da JDK implementan Serializable

Si empregamos unha base de datos relacional para almacenar os obxectos, en lugar de utilizar serialización podemos almacenar os atributos como campos en táboas da base de datos. Habitualmente se empregan librarías de clases deseñadas para xestionar o almacenamento e recuperación de datos denominadas sistemas ***ORM*** (*Object-Relational Mapping*).

Un ***ORM*** é un conxunto de clases que permiten almacenar e recuperar obxectos nunha base de datos relacional de xeito transparente e simple evitando a dependencia de sistemas de bases de datos concretos e incluso sen necesidade de uso das linguaxes de consulta do xestor da base de datos (como SQL).

Un ORM se encarga de almacenar os atributos dos obxectos nas táboas cando "gardamos un obxecto" e de ler os atributos e reconstruír o obxecto cando "lemos un obxecto".
Java proporciona unha API de persistencia para almacenar e recuperar obxectos en bases de datos relacionais denominado ***JPA*** (*Java Persistence API*). A idea de **JPA** é a creación de clases sinxelas denominadas "***entity***" que están asociadas a os rexistros de unha táboa da base de datos relacional. As relacións e correspondencia entre os campos das táboas e os atributos dos obxectos se almacenan mediante ***metadatos*** que poden estar indicados mediante anotacións Java (instrucións de preprocesamento que comezan por @) ou mediante documentos XML independentes que se inclúen coa aplicación.
Existen varias implantacións de JPA de amplo uso como *Hibernate, ObjectDB, TopLink, CocoBase, EclipseLink ou Apache OpenJPA.* Un dos ORM máis empregados en Java é ***Hibernate***. Hibernate é unha implementación de ***JPA*** que pode facer uso de annotations (instruccións de preprocesamento que comezan por @) para asociar clases co almacenamento relacional.

Si as necesidades de almacenamento e recuperación non son tan complexas, tamén se pode facer uso de simples ficheiros no noso sistema de almacenamento principal, en ficheiros secuenciais facendo uso de ***Stream*** de datos ou en ficheiros de acceso aleatorio facendo uso de ***RandomAccessFile***.

Non é imprescindible o uso de ORM para almacenar e recuperar información dunha base de datos relacional, basta dispoñer dunha clase que proporcione unha API de acceso á base de datos dende Java, como JDBC (Java DataBase Connectivity)

Un exemplo de unha clase Hibernate para almacenar obxectos representando a empregados pode ser a seguinte (e importante destacar o uso das anotacións para asociar a clase cunha táboa da base de datos e os atributos cos seus campos)

```
import javax.persistence.*;
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
	@Id @GeneratedValue
	@Column(name = "id")
	private int id;

	@Column(name = "first_name")
	private String firstName;

	@Column(name = "last_name")
	private String lastName;

	@Column(name = "salary")
	private int salary;

	public Employee() {}

	/** Métodos da clase **/
}
```

13.1 – File: O sistema de Arquivos

Os soportes nos dan a posibilidade de almacenar información, pero para poder gardala e recuperala de un xeito eficiente necesitamos organizala dun xeito "comprensible". O xeito en que se organiza a información nun soporte para poder acceder á mesma con facilidade se coñece como ***sistema de arquivos***. Sistemas de arquivos de uso común son FAT32, VFAT, NTFS, EXT4, ZFS, BTRFS ... etc, cada un deles con diferentes características e limitacións. Sen embargo, teñen algo en común: Almacenan a información en Ficheiros (ou Arquivos) e a organizan en Directorios (ou "carpetas").

> O concepto de ***carpeta*** non é máis que unha metáfora para referirnos a un lugar no que agrupamos un conxunto de arquivos, e se coñece así porque nas contornas gráficas de usuario se representan coa icona dunha carpeta. O concepto se denomina ***directorio***

Un arquivo é o lugar onde almacenamos información en formato binario pertencente a un "concepto" concreto. Por exemplo podemos ter información dunha factura, un documento de texto, un vídeo ou unha fotografía.

Podemos distinguir entre dous tipos de arquivos diferentes:

- ***arquivos de texto***: Nos arquivos de texto a información binaria almacenada se interpreta como caracteres *(nun arquivo informático e nos sistemas informáticos toda a información está sempre en binario, e decir representando unicamente valores 0 e valores 1)*. Estes caracteres poden ser interpretados segundo varias codificacións (ISO, UTF-8, UTF-16) e algúns deles teñen un significado "especial" (como o salto de liña). Para interpretar correctamente un arquivo de texto é necesario coñecer que codificación de caracteres utiliza.
- ***arquivos binarios***: Nos arquivos binarios a información non ten unha interpretación concreta. Son arquivos que conteñen bits que poden representar absolutamente calquera cousa: Un programa, unha canción, unha fotografía, ... etc. O modo en que se almacena a información se coñece como "***formato***" do arquivo. Formatos comúns son: class, EXE, PDF, MP4, AVI, MP3, OGG, DOC, MKV, ... etc. Habitualmente para que unha persoa recoñeza de inmediato o tipo de arquivo (o seu formato) se utiliza unha "extensión" para o nome do arquivo que consiste nun punto seguido de 3 letras. Os programas poden ter en conta esa extensión ou non facelo.

Os sistemas de arquivos organizan os arquivos en distintos directorios formando unha árbore na que o raíz ou inicio é o directorio / (\ en Windows). A partir de ahí se van creando directorios e subdirectorios formando unha árbore do tamaño que se desexe. En todos os directorios existen dous directorios "especiais". O directorio **.** que representa a posición actual, e o directorio **..** que representa o directorio anterior ao actual. Para referirnos a un directorio ou arquivo do sistema concreto necesitamos especificar o seu camiño ou ruta (**path**). Un ***path*** pode ser:

- ***Relativo***: Se van listando os directorios polos que necesitamos pasar ata alcanzar o destino contando dende a nosa posición actual. Se separará un directorio de outro utilizando o caracter / si utilizamos un sistema UNIX/Linux ou \ si utilizamos un sistema Windows. Recordemos que **..** representa o directorio anterior ó actual.

- **Absoluto**: Se van listando os directorios polos que necesitamos pasar ata alcanzar o destino contando dende o raíz (e polo tanto o path absoluto sempre comeza por / ou por \ si estamos en Windows).

En Windows existe un concepto que non se utiliza nos sistemas Linux/Unix, que é o concepto de "***unidade***". Baixo Windows o espazo de almacenamento está distribuído en unha ou varias unidades que representan un dispositivo de almacenamento e se nomean cunha letra. Para crear o path é necesario antepoñer a letra da unidade seguida de dous puntos: Por exemplo *C:\Documentos\Java\Titorial.doc* sería o camiño ao ficheiro Titorial.doc dende a carpeta raíz (path absoluto) da unidade C:. Nos sistemas UNIX/Linux non se utiliza o concepto de "unidade", polo que o camiño sería */Documentos/Java/Titorial.doc*

Tamén é importante o concepto de URI (Uniform Resource Identifier) e de URL (Uniform Resource Locator). Unha URI identifica un recurso de xeito que o podemos distinguir de outros recursos. Un camiño a un ficheiro, por exemplo, é unha URI. Existen dous tipos de URI:

- ***URN***: Os URN son "persistentes" no sentido que sempre identificarán o mesmo recurso.
- ***URL***: Os URL non son necesariamente "persistentes" e ademáis de identificar un recurso proporcionan información sobre como acceder ao mesmo normalmente indicando o protocolo de acceso antes do identificador (file://, http:// ftp://)

En Java se da soporte a xestión dos sistemas de arquivos mediante a clase ***File.*** Esta clase proporciona todos os métodos necesarios para determinar e cambiar os permisos dos arquivos, obter o seu *path*, eliminar arquivos e carpetas, listar arquivos, crear carpetas, ver as propiedades dos arquivos... etc.

13.2 – RandomAccessFile

A clase ***RandomAccessFile*** nos permite crear, almacenar e leer información en arquivos sobre un soporte almacenamento de acceso aleatorio.

Antes de poder operar cun ficheiro no disco é necesario “abrilo”, e tamén é imperativo pechalo cando se remate de traballar con él. **En Java o ficheiro se “abre” cando instanciamos o obxecto RandomAccessFile que o referencia**.

A clase RandomAccessFile implementa a interfaz AutoCloseable, polo que pode ser utilizado mediante un try with resources.

O concepto máis importante dos *RandomAccessFile* é o “***punteiro do ficheiro***” ou “posición”. Os bytes almacenados nun ficheiro están identificados por unha dirección, sendo 0 a dirección do primeiro byte, 1 a do segundo... etc. Cando abrimos un *RandomAccessFile* se crea un punteiro do ficheiro que apunta ao byte 0 do mesmo. Cando lemos ou escribimos esta posición vai cambiando de xeito automático para apuntar ó byte sobre o que se realizará a seguinte operación. A clase *RandomAccessFile* dispón de dous métodos para xestionar o posicionamento do punteiro do ficheiro:

- ***long getFilePointer();*** que nos devolve o número do byte ao que está apuntando actualmente o punteiro do ficheiro, e polo tanto o byte que se vai a ler ou escribir na seguinte operación.
- ***void seek(long pos);*** que cambia a posición do punteiro do ficheiro ao byte identificado polo número indicado.

O punteiro do ficheiro almacena a posición do ficheiro onde se realizará a seguinte operación de lectura ou escritura

Os ***RandomAccessFile*** permiten realizar operacións que non son posibles en ficheiros secuenciais como “marcar” rexistros como borrados, reutilizar o espazo marcado como borrado, ou modificar o contido dun rexistro sen necesidade de facer copia do arquivo.

Todos os ficheiros e fluxos de datos deben ser pechados unha vez non precisamos ler ou escribir máis en eles, normalmente para asegurar o peche se recurre ao uso de unha estrutura ***try {...} finally {...}*** que asegura que o bloque **finally** se leve a cabo pase o que pase no método.
Esta aproximación sin embargo plantexa un inconvinte:

```
try {
      RandomAccessFile ras=new RandomAccessFile(“File.dat”);
      /** Traballo co obxecto ras que referencia o ficheiro **/
} finally {
      ras.close();
}
```

*Este código **é erróneo** porque a variable **ras** unicamente ten existencia no bloque delimitado por {} no que foi definida. Polo tanto, no bloque finally {} non existe.*

```
RandomAccessFile ras=new RandomAccessFile("File.dat");
try {
      /** Traballo co obxecto ras que referencia o ficheiro **/
} finally {
      ras.close();
}
try (RandomAccessFile ras=new RandomAccessFile("File.dat")) {
      /** Traballo co obxecto ras que referencia o ficheiro **/
} finally {
      ras.close();
}
```

Este código é correcto. Pero si a apertura do ficheiro falla se levará a cabo todos os xeitos o bloque finally, e se intentará pechar un obxecto que non existe provocando un erro

try with resources. Podemos situar como resources dun bloque try calquera obxecto que implemente a interface Closeable. Iso garante que o obxecto é pechado correctamente pase o que pase durante a execución do método.

13.3 - Fluxos de Datos: Streams

Un ***fluxo de datos*** ou ***Stream*** fai referencia a circulación de información dende un orixe a un destiño. Desde o punto de vista da orixe dos datos o **fluxo é de sáida** *(OutputStream)*, mentres que dende o punto de vista do destiño dos datos é un f**luxo de entrada** *(InputStream)* de datos.
A orixe e destino dos datos pode ser variada: Unha conexión de rede, un ficheiro en disco, unha cámara/tarxeta de vídeo, unha tarxeta de son....etc.

> Un ***Stream*** é un fluxo de datos secuencial. Os datos van dende o orixe ao destino e deben ser leidos segundo chegan, non é posible "avanzar", "posicionarse" ou "retroceder" no fluxo.

Dentro dos ***Stream*** podemos distinguir entre **Streams binarios** (*InputStream* e *OutputStream*) e **Streams de texto** (*Reader* e *Writer*).

13.3.1 – Streams Binarios: InputStream e OutputStream

> Os Streams binarios representan fluxos secuenciais de bytes dende un orixe a un destino.
>
> ***InputStream*** é o tipo de fluxo de entrada binario básico. Representa un fluxo de datos binarios do que podemos ler bytes.
>
> ***OutputStream*** é o tipo de fluxo de saída binario básico. Representa un fluxo de datos binarios no que podemos escribir bytes.

Estas dúas clases proporcionan unha funcionalidade básica polo que se utilizan xunto con clases que ofrecen funcionalidades máis avanzadas. As seguintes clases permiten crear InputStream e OutputStream que teñen distintos orixes e destinos de datos:

- ***ByteArrayInputStream***: Crea un *InputStream* a partir dun array de byte (byte[]) pasado como parámetro ao seu construtor. Cando lemos deste Stream lemos do array de datos.
- ***ByteArrayOutputStream***: Crea un *OutputStream* de xeito que os bytes que escribimos se almacenan nun array de bytes (byte[]). A clase proporciona o método ***byte[] toByteArray();*** que nos permite recuperar os datos escritos.
- ***FileInputStream***: Crea un *InputStream* a partir dun obxecto **File** que se recibe ou crea no construtor da clase e que referencia a un ficheiro en disco. Cando lemos deste Stream lemos datos do ficheiro.
- ***FileOutputStream***: Crea un *OutputStream* de xeito que os bytes que escribimos se almacenan nun ficheiro referenciado polo obxecto **File** que se recibe ou crea no construtor da clase.

As seguintes clases reciben un InputStream ou OutputStream como parámetro no seu construtor e agregan funcionalidades que nos permiten ler ou escribir datos complexos en lugar de únicamente bytes:

- ***DataInputStream***: Permite a lectura de datos primitivos do Stream como *char, boolean, byte, float, double, ou int*.
- ***DataOutputStream***: Permite volcar datos primitivos como *char, boolean, byte, float, double, ou int* a un fluxo de saída.
- ***ObjectInputStream***: Permite ler obxectos dun fluxo de bytes escritos con ObjectOutputStream.
- ***ObjectOutputStream***: Permite volcar obxectos no Stream. Cando se crea o stream sempre se envía unha cabeceira mediante o método ***void writeStreamHeader()***, o que debemos ter en conta si queremos engadir obxectos ao final de un ficheiro xa existente. Nese caso unha solución e empregar unha clase herdada na que este método non faga nada.

> Os obxectos destas clases se crean pasándolle ao construtor como parámetro un InputStream

13.3.2 – Streams de Texto: Reader e Writer

Os Streams de texto representan fluxos de caracteres codificados en UTF-8

Un ***Reader*** e un obxecto que nos permite obter caracteres dun fluxo de caracteres. Os seus métodos so proporcionan métodos simples que non permiten nin ler un String, polo que é habitual o uso de clases máis avanzadas.

Un ***Writer*** é un obxecto que nos permite enviar caracteres a un fluxo de caracteres. Os seus métodos son tamén sinxelos, pero é posible enviar un String ao Stream mediante o método ***void write(String str);***

As seguintes clases permiten crear obxectos *Reader* e *Writer* que obteñen ou envían fluxos de texto:

- ***FileReader***: Define un obxecto *Reader* asociado a un ficheiro en disco.
- ***FileWriter***: Define un obxecto *Writer* asociado a un ficheiro en disco.
- ***StringReader***: Define un obxecto *Reader* asociado a un String
- ***StringWriter***: Define un obxecto Writer asociado a un String

Co obxecto de darlle maior funcionalidade aos obxectos Reader e Writer, existen clases as que se lle pasa un obxecto Reader / Writer no seu construtor para agregarlle funcionalidades:

- ***BufferedReader***: Esta clase proporciona como principal aportación o método ***String readLine()*** que lee do fluxo de texto unha liña.
- ***BufferedWriter***: Esta clase realmente non proporciona novas funcionalidades importantes sobre as ofrecidas por un *Reader* simple, pero mellora a súa velocidade facendo uso de un *buffer* de caracteres.
- ***PrintWriter***: Esta clase proporciona métodos que nos permiten dar formato a conversión en texto da información que queremos escribir no Stream, destacando as distintas versións sobrecargadas dos métodos ***print, println e printf***

Cando visualizamos información na consola con System.out.println, estamos facendo uso do obxecto estático ***out*** que é un atributo da clase System da clase PrintWriter asociada coa consola de texto onde se executa o programa.

13.3.3 – Conversión de Fluxos

Realmente os Stream son sempre fluxos de bytes, non de caracteres ou texto. Os caracteres ou texto non son mais que un xeito específico de interpretar eses bytes. Java proporciona dúas clases que nos permiten crear fluxos de texto a partir de fluxos binarios:

- ***InputStreamReader***: Crea un Reader a partir do InputStream que se lle facilita como parámetro ao seu construtor.
- ***OutputStreamWriter***: Crea un Writer a partir do OutputStream que se lle facilita como parámetro ao seu construtor

Ademais do uso do obxecto Scanner, tamén podemos aceptar datos do teclado creando un obxecto BufferedReader asociado ao obxecto estático ***in***, que é un atributo da clase System da clase *InputStream* asociado ao teclado do sistema. Este obxecto BufferedReader nos permitiría ler liñas do teclado mediante o seu método readLine().

```
BufferedReader reader=new BufferedReader(new InputStreamReader(System.in));
```

13.3.4 – A interface Stream

A interface ***Stream*** define métodos que permiten facer transformacións sucesivas sobre un Stream de datos mediante os métodos proporcionados. A Orixe típica dos datos é unha *Collection* (mediante o seu método ***Stream<E> stream()***) ou un Array (***Arrays.stream***). Mediante os métodos proporcionados por esta interface é posible realizar operacións sobre cada un dos elementos sen necesidade de bucles “for” ou “while”. Estas operacións poden clasificarse en:

- ***Operacións intermedias:*** Unha vez realizado o seu cometido *retornan un novo* ***Stream*** que se pode utilizar para a operación seguinte. Por exemplo *Stream<T> distinct()* que elimina do fluxo os obxectos duplicados.
- ***Operacións terminais***: Producen un resultado final ou realizan unha acción. Por exemplo void forEach(Consumer<? super T> action) que procesa cada un dos elementos do Stream mediante o obxecto action pasado como parámetro.

Os datos dun stream poden ir pasando por unha ou máis operacións intermedias ata acadar unha operación terminal. Este conxunto de accións se coñece como “*stream pipeline*”

Mentres que a clase Stream representa fluxos de obxectos (dos obxectos almacenados na Collection), para Streams de datos primitivos existen as interfaces ***IntStream***. ***DoubleStream*** e ***LongStream*** que nos proporcionan métodos útiles para traballar con Stream numéricos como calcular a suma, o valor máximo, ordeación ....

Mediante os métodos *mapToInt, mapToLong e mapToDouble* de Stream se poden transformar Streams de obxectos a Streams de *int, long ou double*

> Se poden crear Stream mediante métodos estáticos da clase Stream, como of , generate, iterate ou builder.

```
int[] edades={10,7,21,5,19,8,3};
System.out.println(Arrays.stream(edades).max().getAsInt());
```

Visualiza o número maior no array edades

13.4 – Acceso a Sistemas Xestores de bases de datos: JDBC

13.4.1 – Bases de Datos Orientadas a Obxectos

> Un Sistema de ***Base de Datos Orientada a Obxectos*** permite almacenar, recuperar, modificar e eliminar Obxectos dun sistema de almacenamento.

Aínda que no seu día se pensou que este tipo de base de datos se empregaría masivamente realmente non é así, sendo empregados maioritariamente sistemas de bases de datos relacionais como Oracle ou PostgreSQL que hoxe en día incorporan conceptos de orientación a obxectos (Bases de Datos Obxecto-Relacionais). A maior parte das bases de datos relacionais comerciais son de este tipo. Exemplos de Xestores de Bases de Datos Orientadas a Obxectos son *Intersystems Caché, DB4O, Versant Object Database, ObjectStore...*

Neste tipo de base de datos é necesario definir as clases que imos a xestionar mediante a linguaxe propia da base de datos. Unha vez definidas podemos almacenar ,buscar, modificar e borrar ***instancias*** (obxectos) na base de datos.

As bases de datos orientadas a obxectos tamén permiten a execución de métodos dentro da propia base de datos (de xeito similar aos procedementos almacenados ou *trigger*s en SQL). Estes métodos poden estar programados en diversas linguaxes que dependen do xestor utilizado, como *PL/SQL*, *Java*, ou *ObjectScript*. Para a creación de aplicacións complexas existen enlaces (*bindings*) con diversas linguaxes de programación orientadas a obxecto como **Java** que permiten manipular os obxectos almacenados na base de datos dun xeito máis flexible. Por exemplo no xestor *Intersystems Caché* o subsistema ***Caché Java Binding*** permite a creación de clases Java a partir das definicións na base de datos de xeito automático para facilitar o uso dos obxectos dende a aplicación.

13.4.2 – Bases de Datos Relacionais

O método máis habitual de proporcionar persistencia as aplicacións e o uso de bases de datos relacionais. En aplicacións grandes se utilizan habitual sistemas ORM (Object Relational Mapping) baseados en JPA como *Hibernate*, pero en aplicacións máis simples e habitual o acceso e xestión directa da base de datos mediante ***JDBC (Java Database Connectivity)***.

> ***JDBC*** proporciona un conxunto de API que nos permiten realizar as operacións máis comúns sobre bases de datos relacionais sen preocuparnos do xestor concreto que estamos a utilizar, simplemente teremos que utilizar o obxecto que implemente o API máis apropiado para o noso xestor de bases de datos (driver).

Os elementos da API JDBC máis importantes son a clase ***DriverManager*** e a interface ***Connection*** que define os métodos necesarios para traballar coa base de datos (consultar, eliminar, crear táboas.... etc). A clase ***DriverManager*** nos permite cargar a clase que define o obxecto ***Connection*** apropiado para o tipo de base de datos que queiramos utilizar (*driver*) e crear unha instancia co seu método estático ***getConnection()***.

Son tamén imprescindibles as interfaces ***Statement*** e ***PreparedStatement***. Estas interfaces definen os métodos necesarios para realizar operacións coa BBDD como consultas, actualizacións ...etc.

> ***PreparedStatement*** nos permite especificar parámetros en lugar de datos concretos para a realización de consultas, actualizacións ou insercións, que son substituídas polos datos reais filtrados antes do seu proceso evitando inxeccións SQL e acelerando a execución en sentencias repetidas nas que unicamente cambian os parámetros.

Cando realizamos unha operación de consulta, obtemos como resposta un obxecto ***ResultSet***. ***ResultSet*** define os métodos necesarios para recuperar os resultados das consultas.

O seguinte exemplo amosa como realizar unha consulta sobre a táboa "*Equipamento*" de unha base de datos relacional. O proceso de conexión está "ampliado" para que se reutilice con facilidade e de xeito transparente o obxecto *Connection* en lugar de crear un novo para cada operación.

```
public class Database implements AutoCloseable {    // Implementar AutoCloseable permite usar os obxectos con try with resources
    private static final String SERVER="192.168.122.5";
    private static final int PORT=3306;
    private static final String USER="myuser";
    private static final String PASSWORD="test";
    private static final String DB="mydatabase";

    private String url_connection=null;          // String de conexión á base de datos.
    private Connection conn=null;                // Obxecto Connection que implementa o API JDBC para traballar coa BBDD

    /**
     * Construtor
     */
    public Database() {
        try {
            Class.forName("com.mysql.jdbc.Driver");       // Cargamos o obxecto que implementa o API JDBC para mysql
            url_connection = "jdbc:mysql://"+SERVER+":"+PORT+"/"+DB+"user="+USER+"&password="+PASSWORD;
        } catch (SQLException e) {
            System.out.println("SQL Exception: "+ e.toString());
            System.exit(0);
       } catch (ClassNotFoundException cE) {
            System.out.println("Excepción: "+ cE.toString());
            System.exit(0);
        }
    }

    /**
     * Este método devolve o obxecto Connection que permite traballar coa BBDD, asegurándose de que unicamente se cree un obxecto
     * O facemos así, porque si chamamos a getConnection() varias veces sin pechar a ultima conexión en lugar de crear unha conexión nova
     * retornará a conexión existente
     * @return Obxecto Connection coa conexión á nosa BBDD
     */
    public Connection getConnection()  throws SQLException {
        if (conn == null)
          conn = DriverManager.getConnection(cs);  // Conectamos coa BBDD obtendo un obxecto Connection que nos permitirá interactuar con ela
        return conn;
    }

    /**
     * Implementación de AutoCloseable
     */
    @Override
    void close() throws Exception {
         if (conn!=null) {
             conn.close();
             conn=null;
        }
    }
}
```

```
    Database db=new Database();
    try (Connection conn=db.getConnection()) {  // Try with resources
            Statement st=conn.createStatement();
            ResultSet rs=st.executeQuery("SELECT * FROM Equipamento");
            while(rs.next()) {
              System.out.println(rs.getString("descripcion"));
                  System.out.println(rs.getString("ip"));
            }
    } catch(SQLException ex) {
        System.out.println("Error cargando  equipamento ");
        System.exit(0);
    }
```

14. - Programación Concorrente : Thread e Runnable

Un programa é unha secuencia de instrucións que se levan a cabo unha tras outra unha a unha ata chegar a última. Sin embargo, a veces nos interesa que o programa poda realizar varias accións ao mesmo tempo. A isto o chamamos concorrencia ou multiprogramación.

> A baixo nivel a concorrencia real so é posible si o sistema ten varias CPU ou unha CPU con varios cores ou fíos de execución. Si non e así, a concorrencia a ten que xestionar completamente o sistema operativo mediante o seu sistema de planificación.

Existen dous xeitos nos que as aplicacións poden lograr a concurrencia: A creación de procesos (fork) e a creación de fíos de execución (threads).
Na creación de ***procesos***, a aplicación lle indica ao sistema que cree un novo proceso idéntico ao actual denominado "proceso fillo". A partir de ahí o algoritmo pode facer unha cousa si detecta que é o proceso fillo e outra distinta si é o proceso pai. Son dous procesos **independentes** funcionando ao mesmo tempo.
Na creación de ***fíos***, a aplicación executa unha parte do seu código en paralelo co resto (normalmente un método/función). ***Java unicamente soporta fíos, non procesos.***

> O principal problema da programación con fíos é o acceso a recursos comúns a varios fíos. Si ambos fíos intentan modificar ese recurso ao mesmo tempo poden darse inconsistencias graves. A solución é o uso de bloqueos que en Java se obtén mediante a palabra reservada **synchronized**

14.1 – Creación de Fíos de Execución: A interface Runnable.

A clase Java para a creación de fíos e ***Thread***.

> Un obxecto thread executará en paralelo co fío actual o seu método ***public void run();*** cando chamemos ao seu método ***public void start()***

```
class Counter extends Thread {
        int limit;

        public Counter(int limit) {
                this.limit=limit;
        }

        public void run() {
                for(int i=0;i<limit;i++)
                        System.out.println("->"+i);
        }
}
```

```
Counter c=new Counter(25);
c.start();
System.out.println("Continue....");
```

O código continúa despois de start() ao mesmo tempo que se leva a cabo o método run do Counter.

Outra posibilidade é o uso dun obxecto que implemente a interface ***Runnable***. A interface *Runnable* é unha interface funcional que define o método ***public void run();*** e como tal se pode utilizar definindo unha clase normal, unha clase anónima ou unha expresión lambda.

```
class Counter extends Runnable {
        int limit;

        public Counter(int limit) {
                this.limit=limit;
        }

        public void run() {
                for(int i=0;i<limit;i++)
                        System.out.println("->"+i);
        }
}
```

```
Counter c=new Counter(25);
Thread fio=new Thread(c);
fio.start();
System.out.println("Continue....");
```

```
Thread fio=new Thread(new Runnable() {
        public void run() {
                for(int i=0;i<25;i++)
                        System.out.println("->"+i);
        }
});
fio.start();
System.out.println("Continue....");
```

```
Thread fio=new Thread(() -> {
        for(int i=0;i<25;i++)
                System.out.println("->"+i);
});
fio.start();
System.out.println("Continuando....");
```

14.2 – O problema da concorrencia: Sincronización

Si dous fíos intentan acceder ao mesmo recurso ao mesmo tempo poden darse numerosos problemas.

```
public class Conta {
    public int value;

    public Conta(int value) {
        this.value=value;
    }

    public void dec() {
        System.out.println("Decrementando..");
        for(int i=0;i<1000;i++);
        value--;
        System.out.println(value);
    }

    public int val() {
        return value;
    }
}
```

```
public class SyncExample {
    public static void main(String[] args) {
        Conta counter=new Conta(1);

        Thread one=new Thread(()->{
            if (counter.val()>0) counter.dec();
        });

        Thread two=new Thread(()->{
            if (counter.val()>0) counter.dec();
        });

        one.start();
        two.start();
    }
}
```

```
Decrementando..
0
Decrementando..
-1
```

Como podemos observar pode darse o caso de que os dous fíos executen o método *dec()* a pesar de que indicamos que unicamente debe facelo si o ***value*** é > 0. O problema e que os dous fios entran en *dec()* prácticamente ao mesmo tempo antes de que ningún dos dous decremente o valor de ***value***.

A solución sería simplemente declarar os método *dec()* e *val()* como synchronized: ***public synchronized void dec()*** e ***public synchronized int val();***

> Os bloqueos e a xestión da concurrencia é un asunto complexo e a principal dificultade na programación concorrente.

15.- Comunicación TCP/IP Cliente Servidor

O protocolo TCP/IP é o protocolo máis utilizado para a comunicación hoxe en día, sendo a base na que se asenta Internet. Java ofrece soporte para a creación sinxela de programas que se comuniquen mediante este protocolo.
A comunicación TCP/IP na maioría dos casos é de tipo cliente/servidor na que un servizo pasivo (*servidor*) espera conexións por parte de un ou varios *clientes* atendendo e respondendo as súas peticións. Para establecer a comunicación o cliente debe conectarse ao equipo que ofrece o servizo e facer a solicitude de servizo desexada. Para que a comunicación teña lugar será necesario:

1. Identificar o equipo que ofrece o servizo
2. Identificar o programa concreto que ofrece o servizo dentro do equipo

Nunha rede TCP/IP, os equipos se identifican pola súa ***dirección IP***, ou por un nome si dispoñemos dun servizo DNS, mentres que o programa co que desexa contactar o cliente se identifica por un ***número de porto***.
Cando un programa espera solicitudes por parte dun cliente, está a espera de conexións identificándose cun número entre 1 e 65535, denominado *porto*. E habitual que os distintos tipos de servizo utilicen sempre os mesmos números de porto, pero non é obrigatorio. Por exemplo, o servizo ***http*** utiliza habitualmente o porto 80.

> Os números de porto por debaixo do 1024 son números privilexiados. Unicamente o administrador do sistema pode arrancar programas que esperen conexións en números de porto privilexiados.

15.1 - A Resolución de nomes

Recordar números IP non é doado, para simplificar esta tarefa se creou un sistema de resolución de nomes (***DNS***) que se encarga de obter a dirección IP que corresponde ao nome dun equipo. Si non dispoñemos dun servizo ***DNS*** axeitado podemos facer uso do ficheiro hosts, onde mediante liñas asociando a IP co nome podemos proporcionar resolución de nomes, por exemplo: **10.65.21.3 wwww.meuserver.com**

> En Linux este ficheiro se atopa en */etc/hosts* mentres que en Windows está en *C:\Windows\System32\drivers\etc\hosts*

Para xestionar a resolución de nomes, Java dispón da clase ***InetAddress***, que representa unha dirección de internet. Podemos crear un obxecto InetAddress relacionado con www.google.com simplemente así:

```
InetAddress google=InetAddress.getByName("www.google.es");
System.out.println(google.getHostAddress());
```

15.2 - O Servidor: A clase ServerSocket

Un Servidor está a espera (*accept*) de conexións por parte do cliente en unha dirección IP e porto concretos do equipo (*bind*). Cando un cliente establece unha conexión, se obten un obxecto ***Socket*** que permitirá a comunicación entre as dúas partes. O procedemento para a creación do servidor é o seguinte:

1. Creamos un “Socket de Servidor”
2. O vinculamos (bind) a unha IP (ou a todas) e porto concreto do equipo (onde aceptará as peticións)
3. Esperamos a conexión dun cliente.....
4. Cando un cliente se conecta, obtemos un “Socket de Comunicación” que nos permitirá dialogar co cliente ata que decidamos finalizar a conexión

> Si queremos que o servidor poda atender a varios clientes ao mesmo tempo, será necesario lanzar un fío de execución (*Thread*) cada cliente que estableza unha conexión.

Para facer estas tarefas en Java empregamos a clase ***ServerSocket***.

> Unha vez rematada a conexión se debe pechar co método **close()** da clase Socket

O seguinte exemplo amosa un servidor de eco (un servidor que retorna a mesma información que lle envía o cliente) con un so fío (co que será capaz de atender os clientes so de un en un) e con varios fíos (será capaz de atender a múltiples clientes ao mesmo tempo)

Cliente

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;

public class EchoClient {
   public static void main(String[] args) {
      final int PORT=17500;
      String str;
      Scanner scn=new Scanner(System.in);

      try {
         InetAddress server=InetAddress.getByName("localhost");
         try (Socket socket=new Socket(server,PORT)) {   // Try with resources
            BufferedReader reader=new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter writer=new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
            do {
               System.out.print("Texto: ");
               str=scn.nextLine();
               writer.println(str);
               writer.flush();// A escritura se fai nun buffer que non se envía ata que está cheo. Forzamos o envío
               System.out.println("Echo "+reader.readLine());
            } while(!str.equals("bye"));
         }
      } catch(UnknownHostException e) {
         System.out.println("Servidor non atopado");
      } catch(IOException e) {
         System.out.println("Imposible conectar: "+e.getMessage());

      }
   }
}
```

Servidor monotarefa. Atende os clientes de un en un:

```
public class EchoServer {
   private int port;

   private EchoServer(int port) {
      this.port=port;
   }

   private void start() throws IOException {
      // Aceptamos conexións en todas as IP do servidor e o máximo número de clientes a espera de ser atendidos é de 20
      ServerSocket server=new ServerSocket(port,20,null);                    // Esta versión do construtor xa fai o "bind"
      while(true) { // Bucle infinito. O servidor se executa "para sempre"
         String text;

         System.out.println("Esperando conexión de cliente...");
         Socket connection=server.accept();  // Esperamos a conexión dun cliente
         System.out.println("Cliente conectado, lendo e procesando petición...");
         BufferedReader reader=new BufferedReader(new InputStreamReader(connection.getInputStream())); // Fluxo de datos de entrada
         PrintWriter writer=new PrintWriter(new OutputStreamWriter(connection.getOutputStream())); // Fluxo de datos de saida
         do {
            text=reader.readLine();
            System.out.println("Recibido: "+text);
            writer.println(text);
            writer.flush(); // A escritura se fai nun buffer que non se envía ata que está cheo. Forzamos o envío
         } while(!text.equals("bye")); // Bye desconecta o cliente
         connection.close();
      }
   }

   public static void main(String[] args) {
      final int PORT=17500;
      try {
         EchoServer server=new EchoServer(PORT);
         server.start();
      } catch(IOException e) {
         System.out.println("Erro iniciando servidor: "+e.getMessage());
      }
   }
}
```

Servidor multitarefa: Atende varios clientes en paralelo

- Código que atende ao cliente nun fío de execución propio

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;

public class Echo implements Runnable {
   private final Socket connection;

   Echo(Socket connection) {
      this.connection=connection;
   }
   public void run() {
      String text;
      try(connection) {  // try with resources para asegurar o peche do socket
         BufferedReader reader=new BufferedReader(new InputStreamReader(connection.getInputStream())); // Fluxo de datos de entrada
         PrintWriter writer=new PrintWriter(new OutputStreamWriter(connection.getOutputStream())); // Fluxo de datos de saída
         do {
            text=reader.readLine();
            System.out.println("Recibido: "+text);
            writer.println(text);
            writer.flush(); // A escritura se fai nun buffer que non se envía ata que está cheo. Forzamos o envío
         } while(!text.equals("bye")); // Bye desconecta o cliente
      } catch(IOException e) {
         System.out.println("Erro na comunicación: "+e.getMessage());
      }
   }
}
```

- Servidor multifío

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServerThreads {
   private int port;

   private EchoServerThreads(int port) {
      this.port=port;
   }

   private void start() throws IOException {
      // Aceptamos conexións en todas as IP do servidor e o máximo número de clientes a espera de ser atendidos é de 20
      ServerSocket server=new ServerSocket(port,20,null);        // Esta versión do construtor xa fai o "bind"
      while(true) { // Bucle infinito. O servidor se executa "para sempre"
         String text;

         System.out.println("Esperando conexión de cliente...");
         Socket connection=server.accept();  // Esperamos a conexión dun cliente
         System.out.println("Cliente conectado, lendo e procesando petición...");
         Thread worker=new Thread(new Echo(connection));          // Lanzamos un fío que se encargue de atender ao cliente
         worker.start();
      }
   }

   public static void main(String[] args) {
      final int PORT=17500;
      try {
         EchoServerThreads server=new EchoServerThreads(PORT);
         server.start();
      } catch(IOException e) {
         System.out.println("Erro iniciando servidor: "+e.getMessage());
      }
   }
}
```

16.- Programación de GUI: JavaFX

O fluxo habitual de execución dunha aplicación en un contorno textual é completamente secuencial: A aplicación comeza na primeira instrución e se vai executando secuencialmente ata rematar coa última. As instrucións poden ser expresións matemáticas, lóxicas, chamadas a funcións, iteracións ou execucións condicionais (selección). Dentro das chamadas as funcións podemos distinguir as operacións de entrada de datos por teclado e a saída de datos pola pantalla.

En particular, cando o programa atopa unha chamada que espera que o usuario introduza un dato se chama a unha función que bloquea a secuencia de execución ata que o usuario realiza a entrada, tras o que continúa o proceso.

Cando creamos unha aplicación que interactúa co usuario mediante unha interface gráfica o fluxo de execución cambia completamente. Nos contornos gráficos, todas as aplicacións están permanentemente executando un bucle de xestión de eventos no que se recollen os eventos que se producen e se envían ao xestor de eventos apropiado que se encargará de realizar as operacións desexadas. Si polo que fora a xestión do evento tarda moito en realizarse ou non retorna o control ao bucle de xestión de eventos, o contorno gráfico da aplicación quedará conxelado impedindo a interacción.

> Un programa con contorno gráfico sempre consistira na acción de recoller-evento, despachar-evento mentres no se reciba un evento de “finalización do programa”.

Para realizar un programa que interactúe co usuario mediante unha GUI é necesario polo tanto deseñar a propia GUI cos distintos elementos colocados de xeito apropiado (etiquetas, campos de entrada de datos, botóns, radio buttons, check boxes, campos de selección, menús, campos de edición .... etc) e logo programar os xestores de eventos que se encarguen de xestionar as distintas accións que o usuario poda realizar sobre os mesmos (pulsar botóns, elexir un elemento dun menú ou dunha caixa de selección, marchar un check-box .... etc). Este tipo de programación se denomina ***programación orientada a eventos***.

16.1 - A Programación Visual

A programación orientada a eventos é pesada e laboriosa. En particular, a codificación do aspecto do GUI da aplicación (definición e colocación dos distintos elementos gráficos) e longa e aburrida de codificar. Para facilitar a programación de esta parte das aplicacións se empregan *deseñadores de GUI.*
Un *deseñador de GUI* é unha aplicación que nos ofrece unha paleta de distintos elementos gráficos que podemos elixir e colocar para facer o noso deseño evitando a necesidade de codificación manual do aspecto da aplicación. Os deseñadores tamén nos ofrecen a miúdo a posibilidade de indicar os eventos dos distintos elementos que desexamos xestionar indicando o método concreto que se encargará de facelo. Dese xeito, a programación da aplicación é basicamente o deseño da aplicación arrastrando e colocando elementos gráficos e a codificación dos métodos de xestión dos eventos pertinentes. Este tipo de programación se denomina ***programación visual***

16.2 - Principais Elementos da GUI, Eventos e Interfaces Implicadas

Cada elemento que podemos colocar na GUI posúe unha serie de atributos (características) e un conxunto de métodos, como todos os obxectos, pero ademais teñen a capacidade de xestionar unha serie de eventos que “disparan” en resposta as distintas accións do usuario. Para esta xestión teñen como atributos obxectos que implantan a interface ***EventListener***. ***EventListener*** é unha interface que non define métodos, se non que se utiliza simplemente como base para todos os interfaces “Listener” necesarios para os distintos elementos gráficos. Algúns interfaces *EventListener* utilizados de xeito común son:

- ***ActionListener***: define o método ***void actionPerformed(ActionEvent e)***. Este método se dispara cando se realiza unha acción interactuando cun elemento, como por exemplo pulsar un botón. Os obxectos *ActionListener* xestionan eventos de tipo ***ActionEvent***
- ***KeyListener***: define os métodos necesarios para atender aos distintos eventos de pulsación de teclas, como ***void keyPressed(KeyEvent e); void keyReleased(KeyEvent e); void keyTyped(KeyEvent e);.*** Os obxectos *KeyListener* xestionan eventos de tipo ***KeyEvent***
- ***MouseListener***: define os métodos necesarios para atender eventos do rato como v*oid mouseClicked(MouseEvente e) ou mousePressed(MouseEvent e)*. Os obxectos MouseListener xestionan eventos de tipo ***MouseEvent***.

Existen moitos máis *Listener* como *MouseMotionListener* (que xestiona eventos necesarios para o arrastre de elementos), *WindowsListener, FocusListener, TextListener, ItemListener, AdjustmentListener* ....

> Todos os *Listener* reciben como parámetro un obxecto ***Event*** que contén a información concreta do evento producido.

Os distintos obxectos gráficos dispoñen dos métodos necesarios para engadir os "Listener" necesarios segundo os eventos que son capaces de xestionar, usualmente da forma *addActionListener*, *addKeyListener*.... etc, recibindo como parámetro un obxecto que implanta interface apropiada.

16.3 - Conceptos Xerais das GUI: Os administradores de Deseño

Un dos problemas maiores no deseño das GUI é a correcta colocación dos distintos elementos. A dificultade radica en que unha aplicación se vai a executar en distintos dispositivos con distintas resolucións de pantalla, as fiestras poden cambiar o seu tamaño e facerse maiores ou máis pequenas.... etc.

Si colocamos os elementos da nosa interface en posicións fixas, cando a resolución da pantalla non é suficiente algúns elementos é posible que non aparezan e polo tanto non se poda interactuar con eles. Pola contra, si a resolución é maior, veremos os elementos "apiñados" e teremos unha gran área da interface baleira. O mesmo nos ocorre si variamos o tamaño da fiestra.

> É moi aconsellable en xeral non colocar os elementos en posicións fixas, se non en posicións relativas.

Para facilitar o deseño, Java (e moitas outras linguaxes que permiten traballar con GUI) utiliza unha serie de obxectos gráficos que serven para organizar o seu contido de xeito automático readaptándose ao tamaño dispoñible: Os **administradores de deseño** ou ***Layouts***.
A libraría ***Swing*** de Java define os seguintes *Layouts* (que son utilizables dende outras librarías como ***JavaFX***): *AbsoluteLayout, BorderLayout, BoxLayout, CardLayout, FlowLayout, GridBagLayout, GridLayout, OverlayLayout*

16.4 - A arquitectura MVC

En moitas linguaxes de programación o deseñador crea o código na linguaxe de programación empregada para xerar a interface gráfica, permitindo engadir o código que desexamos para cada *Listener* desexado. Esta aproximación visual acelera moito o desenvolvemento pero a interface de usuario é dependente do código e polo tanto é de difícil modificación. Alterar o aspecto dunha aplicación moitas veces exixe unha recodificación a non ser que se teña moito coidado. Por outra banda que o comportamento da aplicación estea ligada a interface de usuario fai menos reutilizable o código.

A aproximación a empregar hoxe en día tanto no deseño de aplicacións con GUI como aplicacións web se coñece como ***Modelo-Vista-Controlador***. Esta aproximación pretende dividir as aplicacións en 3 partes independentes e diferenciadas:

- **Vista**: Se encarga da visualización e entrada de datos. E a interfaz do programa co usuario. No noso caso a GUI
- **Controlador**: Recibe as peticións realizadas a través da vista, solicita a realización da operación ao Modelo recibindo o resultado e envía os resultados de novo a Vista para a ser mostrados.
- **Modelo**: Realiza as operacións do programa. O Modelo nunca solicitará datos ao usuario (os recibe) nin os visualizará (os devolve), simplemente realizará as distintas operacións da aplicación.

Si seguimos este enfoque, o Modelo nos proporcionará un conxunto de obxectos útiles que se poden reutilizar en moitos ámbitos permitindo o desenvolvemento de servizos web, aplicacións de escritorio ou aplicacións web utilizando exactamente o mesmo modelo e precisando basicamente so modificar as vistas.

16.5 - AWT, Swing e JavaFX.

Para crear unha interface gráfica nunha aplicación de escritorio necesitamos:

- As librarías para debuxar no contorno
- O sistema de xestión de eventos
- O toolkit gráfico (consistente nos distintos elementos que queremos pintar nos programas)

En Java a primeira libraría de amplo uso con estas características foi ***AWT*** que hoxe en día non se utiliza directamente.

Swing é unha evolución de *AWT* (utiliza librarías base de AWT) que aporta moitas melloras e facilidade para a creación dos contornos gráficos. Swing foi un avance significativo sobre AWT, pero a interface da aplicación debe ser definida obrigatoriamente mediante código, de xeito que é necesario recompilar para variar o aspecto gráfico. ***JavaFX*** naceu para substituír a *Swing* incorporando moitas características novas. Quizais a máis relevante agora para nos é a posibilidade de definir a interface de usuario mediante un documento XML (.fxml) de xeito que é perfectamente posible cambiar completamente o aspecto gráfico dunha aplicación sen necesidade de programar nin recompilar nada, creando unha descrición XML da GUI. Unha aplicación pode dispoñer de varios XML e decidir en tempo de execución o aspecto que queremos (skins). ***JavaFX*** proporciona tamén a posibilidade de utilizar follas de estilo similares ao CSS, utilidades para a animación
e gráficos 2D, 3D e deseño para móbiles.

Mentres que en Swing as aplicacións non son mais que unha ou varias fiestras cunha serie de compoñentes, en ***JavaFX*** as aplicacións consisten en tres tipos de obxectos:

- ***Stage***: Normalmente é a aplicación. E a parte "externa" e estará composta de unha ou varias ***Scene***.
- ***Scene***: Nun *Stage* se van alternando diversas *Scene* (de xeito similar as **Activities** en Android)
- **Root Pane / Container**: E a interface que conten todos os elementos da aplicación, normalmente organizados mediante algún tipo de *Layout*.

No seguinte enlace podes ver o xeito de habilitar Netbeans para o desenvolvemento de aplicacións JavaFX.

17.- Documentación dos Programas: Javadoc

Javadoc permite xerar documentación sobre o API da aplicación a partir dos comentarios. Para conseguir isto se deben empregar annotations específicas que indican ao analizador javadoc como interpretalos. Annotations típicas de Javadoc son: *@Author, @param, @return, @see, @version,@deprecated,@throws* ...etc

18.- A verificación dos programas: Tests de Unidade con JUnit

18.1 – Introdución a JUnit

As ***probas de unidade*** consisten en comprobar que a clase responde axeitadamente ás distintas entradas de datos proporcionadas. Si o conxunto de datos elixido é apropiado se garantirá o seu correcto funcionamento. Polo tanto, os test de unidade non son mais que "pequenos programas" que van subministrando uns datos de entrada e comprobando que a saída sexa correcta. Os test de unidade tamén poden servir de documentación dos requisitos funcionais (xa que os test reflicten cal debe ser o comportamento da clase) e facilitar o mantemento, xa que unha vez escritos poden volver a executarse cada vez que se modifica algo ou se engaden funcionalidades.

Co obxectivo de facilitar este traballo e sobre todo estandarizalo de xeito que os tests sexan efectivamente parte da documentación aparecen os frameworks de test de unidade como ***JUnit***. ***JUnit*** é un framework para a construción de test Unitarios en Java. Este framework fai uso de anotacións Java (annotations) para identificar os métodos que forman parte dos tests. Un ***test JUnit*** é un método de unha clase que se utiliza unicamente para probas. Esta "*Clase para Testing*" se sinala mediante a *annotation* **@Test**.

> Nos métodos da *clase para testing* se utilizan métodos ***assert*** proporcionados por JUnit ou outro framework, para comprobar que os resultados devoltos polo método a comprobar se corresponden co resultado esperado.

Nas chamadas aos métodos dentro do test deberíamos indicar mensaxes explicativos que permitan identificar correctamente o lugar e motivo do fallo facilitando así a súa corrección, sobre todo tendo en conta que o programador encargado de realizar os tests unitarios non é necesariamente o programador que codificou a clase.

18.2 - Regras de estilo JUnit

Os tests ***JUnit*** utilizan distintas convencións de estilo:

- Se debería utilizar como sufixo ou prefixo a palabra Test, para os nomes das clases, como *TestCliente* ou *ClienteTest*.
- O nome dun test debería ser explicativo, indicando que estamos probando. Si se realiza correctamente non debe ser necesario ler o código para saber que e o que o test está comprobando.
- Se debería utilizar "*should*" nos nomes dos métodos de proba. Por exemplo, *orderClienteShouldBeCreated*, que indicaría o que debería pasar si o método se executa correctamente.

- Se debería facer uso de *Given [explicacion da entrada] When [explicacion do que se fai] Then [resultadoesperado]*. Por exemplo, *givenProductoWhenNotExistenciasThenThrowOutofStockException*

A idea xeral é que mediante a creación de identificadores significativos é posible saber perfectamente o que estamos probando sen necesidade de ler o código das probas.

18.3 - Execución de Test Unitarios JUnit desde o terminal

A clase ***org.junit.runner.JUnitCore*** nos proporciona o método ***runClasses()***. Este método nos permite executar unha ou varias clases de Test JUnit. O valor devolto por este método é un obxecto de tipo ***org.junit.runner.Result***, que podemos utilizar para obter a información sobre os resultados dos tests. O seguinte exemplo indica como levar a cabo o test unitario JUnit correspondente a clase *MyClass* (*MyClassTest*), indicando na consola os posibles fallos:

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class MyTestRunner {
        public static void main(String[] args) {
        Result result = JUnitCore.runClasses(MyClassTest.class);
        for (Failure failure : result.getFailures()) {
                System.out.println(failure.toString());
        }
        }
}
```

Esta clase pode executarse como calquera outro programa Java dende a liña de comandos, aínda que pode ser necesario engadir á variable *CLASSPATH* a localización da librería JUnit ou indicala con **-cp**.

JUnit supón que todos os métodos de test poden executarse nun orden arbitrario. O código de test ben escrito non debe asumir ningún orden, e dicir un test non debe depender de outros. O comportamento por defecto de *JUnit 4* é utilizar un orden aleatorio para a execución dos tests.

E posible utilizar unha *annotation* para definir que os métodos se ordenen lexicograficamente polo seu nome, ou outras variacións:

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
@FixMethodOrder(MethodSorters.DEFAULT)
@FixMethodOrder(MethodSorters.JVM)
```

A seguinte táboa amosa as anotacións máis importantes para utilizar nos métodos en JUnit 4, que deben ser importadas mediante ***import org.junit.<Anotacion>***

Anotación	Descripción
@Test	Identifica o método como un método de Test JUnit
@Before	Este método se executa antes de cada Test
@After	Este método se executa á finalización de cada test
@BeforeClass	Se executa unha vez ao comenzo do Test. O método debe ser static
@AfterClass	Se executa unha vez ao final do Test. O método debe ser static
@Ignore ou @Ignore("Mensaxe")	Indica que o test non está activo.
@Test (expected = Exception.class)	Falla si o método non lanza a Exception indicada.
@Test(timeout=100)	Falla si o métdo tarda máis de 100ms en executarse

18.4 – Creación de Test JUnit

JUnit nos proporciona a clase **Assert**, que dispón de varios métodos *static* para comprobar certas condicións de execución. Tipicamente estes métodos comenzan con ***assert***, permitindo especificar unha mensaxe de erro, o resultado esperado e o resultado actual. O método compara o valor actual co valor esperado, lanzando unha excepción si falla. A seguinte táboa amosa un resumo de estes métodos.

Statement	Description
fail([message])	Fai que o método falle.
assertTrue([message,] boolean condition)	Comproba que a condición sexa certa
assertFalse([message,] boolean condition)	Comproba que a condición sexa falsa
assertEquals([message,] expected, actual)	Comproba que os valores *expected* e *actual* sexan iguais.

assertEquals([message,] expected, actual, tolerance)	Comproba que dous *double* ou *float* son “iguais”, permitindo unha desviación de *tolerance* decimais
assertNull([message,] object)	Comproba que o obxecto é null
assertNotNull([message,] object)	Comproba que o obxecto non é null
assertSame([message,] expected, actual)	Comproba que *expected* e *actual* referencian ao mesmo obxecto
assertNotSame([message,] expected, actual)	Comproba que *expected* e *actual* NON referenacian ao mesmo obxecto

O seguinte exemplo mostra un test unitario para a clase MyClass que comproba o correcto funcionamento do seu método *int multiply(int a,int b)* que retorna o resultado da multiplicación de dous números

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MyTests {
   @Test
   public void multiplicationOfZeroIntegersShouldReturnZero() {
      Multiply tester = new Multiply(); // Clase a probar

      // assert statements
      assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
      assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
      assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
   }
}
```

```
// class to be tested
class Multiply {
      public int multiply(int i, int j) {
         return i *j;
      }
}
```

18.5 – Test Suites JUnit

Si dispoñemos de varias clases de Test, podemos combinalas nunha "*Test Suite*". Executando unha "*Test Suite*", comprobamos todas as clases de test en esa suite na orde especificada. Unha Test Suite pode incluír outras test suites. O seguinte código amosa o uso de unha ***Test Suite***. Contén dúas clases de proba (***MyClassTest*** e ***MySecondClassTest***). Para engadir máis clases de proba bastará con engadilas á anotación *@Suite.SuiteClasses.*

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({
      MyClassTest.class,
      MySecondClassTest.class })

public class AllTests {
}
```

18.6 – Anulando Tests

Si non queremos que se leve a cabo algún test, bastará con engadirlle a annotation ***@Ignore***. Sen embargo, tamén é posible utilizar ***Assume.assumeFalse,*** ou ***Assume.assumeTrue*** dentro do test para que sexa ignorado. Por exemplo:

Assume.assumeFalse(System.getProperty("os.name").contains("Linux"));

anula o test (o marca como test non válido) si o sistema é Linux.

18.7 – Tests Parametrizados.

JUnit permite deseñar tests aos que se lles poden pasar parámetros, o que resulta moito máis cómodo para a realización de tests con gran cantidade de datos. Isto faise marcando a clase de test mediante a *annotation* ***@RunWith(Parameterized.class),*** e creando un método estático coa anotación ***@Parameters*** que retorne unha *Collection* de Arrays. Cada elemento de esa Collection se utilizará como datos para o test.

Para obter valores concretos dos parámetros para cada test, podemos utilizar a *annotation* **@Parameter(n)** nos atributos públicos.

O seguinte test, comproba o funcionamento dunha clase ***Multiply***, que dispón dun método *multiply* que multiplica dous enteiros:

```
import static org.junit.Assert.assertEquals;
import static org.junit.runners.Parameterized.*;

@RunWith(Parameterized.class)
public class ParameterizedTestFields {      // fields used together with @Parameter must be public
   @Parameter(0)
   public int m1;
   @Parameter(1)
   public int m2;
   @Parameter(2)
   public int result;

   // creates the test data
   // Unha Collection é un conxunto de obxectos. En este caso un conxunto de Arrays de tres elementos enteiros
   @Parameters
   public static Collection<Object[]> data() {
      Object[][] data = new Object[][] { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
      return Arrays.asList(data); // transforma o Array de dúas dimensións a unha lista de Arrays
   }

   @Test
   public void testMultiplyException() {
      Multiply tester = new Multiply();
      // De xeito automático, m1 e m2 van tomando todos os valores da lista
      assertEquals("Result", result, tester.multiply(m1, m2));
   }
}
```

```
// class to be tested
class Multiply {
      public int multiply(int i, int j) {
         return i *j;
      }
}
```

Tamén é posible utilizar un construtor na clase de Test para recoller os valores subministrados polo método anotado con @Parameters, en lugar de utilizar a *annotation* ***@Parameter***. Ese construtor debe ter tantos argumentos como os elementos do Array subministrado:

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;

import java.util.Arrays;
import java.util.Collection;

import static org.junit.Assert.assertEquals;
import static org.junit.runners.Parameterized.*;

@RunWith(Parameterized.class)
public class ParameterizedTestFields {
   public int m1;
   public int m2;
   public int result;

   // Constructor para recoller os parámetros do test
  public ParametrizedTestFields(int m1,int m2,int r) {
            this.m1=m1;
            this.m2=m2;
            this.result=r;
  }

   // creates the test data
   // Unha Collection é un conxunto de obxectos. En este caso un conxunto de Arrays de tres elementos enteiros
   @Parameters
   public static Collection<Object[]> data() {
      Object[][] data = new Object[][] { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
      return Arrays.asList(data); // transforma o Array de dúas dimensións a unha lista de Arrays
   }

   @Test
   public void testMultiplyException() {
      Multiply tester = new Multiply();
      // De xeito automático, m1 e m2 van tomando todos os valores da lista
      assertEquals("Result", result, tester.multiply(m1, m2));
   }
}
```

```
// class to be tested
class Multiply {
      public int multiply(int i, int j) {
         return i *j;
      }
}
```

18.8 - Regras JUnit

As regras *JUnit* permiten configurar o comportamento dos tests mediante unha serie de obxectos que definen regras. Estas regras consisten en atributos anotados mediante ***@Rule***, almacenando un obxecto da clase **Rule** (*org.junit.rules*) representando unha regra.
Algúns dos obxectos ***Rule*** subministrados con JUnit son: *TemporaryFolder*, *ExpectedException*, *ExternalResource*, *ErrorCollector*, *Verifier*, *TestWatcher*, *TestName*, *Timeout*....

```
apackage de.vogella.junit.first;

import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class RuleExceptionTesterExample {

  // Creamos o obxecto ExpectedException (o Rule) que nos permitirá verificar as excepcións
  @Rule
  public ExpectedException exception = ExpectedException.none();

  @Test
  public void throwsIllegalArgumentExceptionIfIconIsNull() {
    // O test debe lanzar unha excepción IllegalArgumentException para non fallar
    exception.expect(IllegalArgumentException.class);
   // A mensaxe da excepción debe ser "Negative value not allowed" para non fallar
    exception.expectMessage("Negative value not allowed");
    ClassToBeTested t = new ClassToBeTested();
    t.methodToBeTest(-1);
  }
}
```

18.9 – Desenvolvemento orientado a probas TDD (Test Driven Development)

O ***desenvolvemento dirixido por probas*** consiste en desenvolver a aplicación a partir das probas que verifican que os requisitos esixidos se cumpren. O proceso a seguir é o seguinte:

1. Se elaboran os *casos de proba* a partir dos requisitos esixidos. Eses *casos de proba* deben consistir na elección nun conxunto de datos de entrada seleccionados para verificar a corrección dos algoritmos e as saídas que deben producir. Este tipo de probas nas que non nos interesa a estrutura do algoritmo si non unicamente si o resultado producido é correcto para calquera dato de entrada se denominan **probas de caixa negra**. Existen varias técnicas de elección de datos, como valores límite ou partición de equivalencias.
2. Se crea o proxecto, pero sin escribir código
3. Se elixen os casos de proba deseñados para verificar o cumprimento do primeiro requirimento e se codifican.
4. Se pasa o test a aplicación
5. Si temos fallos, os corriximos, e volvemos ao punto 4
6. Se elixen os casos de proba seguintes mentres non se rematen e se volve ao punto 4

Unha vez rematados estes pasos, o software estará finalizado e será correcto dende o punto de vista dos requirimentos.

18.9.1 – Exemplo: A Clase Complex

Captura de requisitos (especificacións funcionais):

Desexamos desenvolver unha clase que nos permita xestionar números complexos e facer as operacións básicas sobre eles. Acudimos a: https://aga.frba.utn.edu.ar/definicion-y-operaciones-de-numeros-complejos-en-forma-binomica/ para documentarnos e chegamos ás seguintes conclusións:

- Un número complexo se define con dous números reais. O primeiro se chamará "parte real" e o segundo "parte imaxinaria"
- Si un número complexo ten como parte imaxinaria 0, corresponde co número real correspondente ao valor da parte real
- Si un número complexo ten a parte real a 0, se denomina "imaxinario puro"

- Un número complexo se pode representar como un par de valores (a,b), onde ***a*** é a parte real e ***b*** a parte imaxinaria, ou en "forma binómica": ***a+bi***, onde **i** corresponde co valor imaxinario **(0,1)**
- Sexan os números complexos (a,b) e (c,d). As operacións sobre números complexos serán:
  - SUMA:(a,b)+(c,d) = (a+c,b+d)
  - PRODUCTO:(a,b)*(c,d) = (a*c-b*d,a*d+b*c)
  - i^2 = (0,1) * (0,1) = (0*0-1*1, 0*1+1*0) = (-1,0) → i^2 = -1
  - CONXUGADO: o conxugado de (a,b) se define como (a,-b), e cumple as seguintes propiedades:
    - Sexa z un número complexo:
      - z+conxugado(z) = 2* parte real de z
      - z-conxugado(z) = 2*i*parte imaxinaria de z
      - z*conxugado(z) = parte real² +parte imaxinaria²
  - DIVISION: Sexan z1 e z2 números complexos, z1/z2 = (z1 * conxugado(z2)) / (z2 * conxugado(z2)

Elaboración dos casos de proba

Se trata de utilizar os requisitos funcionais expostos arriba para elaborar un suficiente número de casos concretos significativos que describan con precisión en que consisten as operacións. Os casos concretos que van a verificar os requisitos funcionais da clase Complexo son:

- Creación dun número complexo.
  - Si creamos un número Complexo (5.0,-3.0) a parte real é 5.0 e a parte imaxinaria 3.0
  - Si creamos un número Complexo (0.0,1.0) a parte real é 0.0 e a parte imaxinaria é 1.0
- Representación binómica
  - A representación binómica do número Complexo (0.0,1.0) é i
  - A representación binómica do número Complexo (0.0,5.0) é 5i
  - A representación binómica do número Complexo (0.0,-3.0) é -3i
  - A representación binómica do número Complexo (-1.0,0.0) é -1
  - A representación binómica do número Complexo (7.23,0.0) é 7.23
  - A representación binómica do número Complexo (5.127, 4.213) é 5.127+4.213i
  - A representación binómica do número Complexo (3.2,-6.25) é 3.2-6.25i
- Suma:
  - (0.0,1.0) + (1.0,0.0) = (1.0,1.0) ou sexa 1+i
  - (-3.0,8.0) + (5.0,1.0) = (2.0,9.0) ou sexa 2+9i
  - (-1.5,2.0) + (1.5,-2.5) = (0.0,0.5) ou sexa 0+0.5i
- Producto:
  - (0,1)*(0,1)=(-1,0) ou sexa -1
  - (2,-3)*(5,2)=(16,-11) ou sexa 16-11i
  - (0,1)*(8,0)=(0,8) ou sexa 8i
- Conxugado
  - conxugado(5.0,3.0) é (5.0,-3.0)
  - conxugado(0.0,1.0) é (0.0,-1.0)
  - (5.0,3.0)+conxugado(5.0,3.0) = (10.0,0.0)
  - (5.0,3.0)-conxugado(5.0,3.0) = (0.0,6.0)
  - (5.0,3.0)*conxugado(5.0,3.0) = (34.0,0.0)
- División
  - (4.0,3.0) / (4.0,3.0) = (1.0,0.0)
  - (5.0,8.0) / (7.0,-1.0) = (0.5,1.3)
  - (49,35) / (7,0) = (7,5)
  - (3,2) / (-1,2) = (0.2, 1.6)

Creación do Test

Unha vez deseñados os casos de proba se pasaría a codificación dos test JUnit.