Introduccion a Java

Variables

Constantes

`final`

```
final <tipo> <CONSTANTE> = <valor>;
```

donde `<valor>` se escribe: byte: `(byte)64`, short: `(short)64`, int: `64`, long: `64L`, float: `64.0f`, double: `64.0`, char: `'@'` ó `'\u0040'`, boolean: `true` / `false` objetos: **`null`**, String: `"64"`, vectores: `{<valor>, <valor>, …}`

Ejemplo: `final int MAX_ELEM = 20;`

Tipos simple o primitivos

```
<tipo_simple> <variable> [= <valor>];
```

Ejemplo: `int i = 0;`

tipo	tamaño	rango		envolvente
byte	8 bits	-128	127	Byte
short	16 bits	-32.768	32.767	Short
int	32 bits	-2.147.483.648	2.147.483.647	Integer
long	64 bits	$-9.223.372 \cdot 10^{12}$	9.223.372.036.854.775.807	Long
float	32 bits	$-3.4 \cdot 10^{38}$	$3.4 \cdot 10^{38}$ (mínimo $1.4 \cdot 10^{-45}$)	Float
double	64 bits	$-1.8 \cdot 10^{308}$	$1.8 \cdot 10^{308}$ (mínimo $4.9 \cdot 10^{-324}$)	Double
boolean		false	true	Boolean
char	16 bits	Unicode 0	Unicode $2^{16}-1$	Character
void	0 bits	-	-	Void

Tipos compuestos

`new`

```
<tipo_compuesto> <variable> = new <tipo_compuesto>(<param>);
```

Pueden ser: arrays o clases. Dentro de las clases existen algunas especiales: *envolventes*, `String`, *colecciones* y *enumerados*

Son referencias (punteros)

Arrays

```
<tipo><array>[]..[] = new <tipo>[<num>]..[<num>];
```

El primer elemento es el 0, al crearlos (new) hay que saber su tamaño

```
float v[] = new float[10]; //Una dimensión y 10 elementos
float M[][]= new float[3][4]; //Dos dimensiones
String s[] = {"hola", "adios"}; // elementos inicializados
for (int i = 0; i < M.length; i++)
  for (int j = 0; j < M[i].length; j++)
    M[i][j] = 0;
```

Envolventes (wrappers)

Para cada uno de los tipos simples existe una clase equivalente, con constantes y métodos que nos pueden ser útiles. Ver tabla en *variable simple*. Ver *conversión de tipos*.

Cadena caracteres

`String`

```
String <nombre_variable> [= "<cadena de caracteres>"];
```

Ejemplo: `String s = "Hola";` ó `String s = new String("Hola");`

Métodos de la clase `String`:

```
.equals(String s2) //compara dos Strings
```

```
.clone() //crea una copia de un String
.charAt(int pos) //retorna el carácter en una posición
.concat(String s2) //concatena con otro Strings
.indexOf(char c, int pos) //devuelve posición de un carácter
.length() //devuelve la longitud del String
.replace(char c1, char c2) // reemplaza un carácter por otro
.substring(int pos1, int pos2) // extrae una porción del string
.toLowerCase() // convierte el String a minúsculas
.toUpperCase() // convierte el String a mayúsculas
.valueOf(int/float/… numero)  // convierte un número a String
```

Colecciones

El API de Java nos proporciona colecciones donde guardar series de datos de cualquier clase. Dichas colecciones no forman parte del lenguaje, sino que son clases definidas en el paquete *java.util*.

```
<Tipo_colecc><<Clase>> <colección> = new <Tipo_colecc><<Clase>>();
```

Hay tres tipos, cada uno con un interfaz común y diferentes implementaciones:

Hay tres tipos, cada uno con una interfaz común y diferentes implementaciones:

Listas – interfaz: `List<E>` Estructura secuencial, donde cada elemento tiene un índice o posición: `ArrayList<E>` (acceso rápido), `LinkedList<E>` (inserción/borrado rápido), `Stack<E>` (pila), `Vector<E>` (obsoleto).

Conjunto – interfaz: `Set<E>` Los elementos no tienen un orden y no se permiten duplicados: `HashSet<E>` (la implementación usa tabla *hash*), `LinkedHashSet<E>` (+ doble lista enlazada), `TreeSet<E>` (usa árbol).

Diccionario o Matriz asociativa – interfaz: `Map<K,V>` Cada elemento tiene asociada una clave que usaremos para recuperarlo (en lugar del índice de un vector): `HashMap<K,V>`, `TreeMap<K,V>`, `LinkedHashMap<K,V>`

Las interfaces `Iterator` y `ListIterator` facilitan recorrer colecciones. La clase estática `Collections` nos ofrece herramientas para ordenar y buscar en colecciones..

```
ArrayList<Complejo> lista = new ArrayList<Complejo>();
lista.add( new Complejo(1.0, 5.0) );
lista.add( new Complejo(2.0, 4.2) );
lista.add( new Complejo(3.0, 0.0) );
for (Complejo c: lista) {
  System.out.println( c.getNombre() );
}
```

Enumerados

Enum

```
enum <nombre_enumeracion> { <CONSTANTE>, ..., < CONSTANTE> }
```

Ejemplo:

```
enum estacion { PRIMAVERA, VERANO, OTOÑO, INVIERNO };
        estacion a = estacion.VERANO;
```

Ámbito

Indica la vida de una variable, se determina por la ubicación de llaves { } donde se ha definido.

```
{
  int  a  =  10;  // sólo a  disponible
  {
    int b =  20; // a y b disponibles
  }
  //  sólo a  disponible
}
```

Expresiones y sentencias

Comentarios

```
// Comentario de una línea
/* Comentario de
   varias líneas */
/** Comentario javadoc: para crear automáticamente la
    documentación de tu clase */
```

Operadores

asignación: `=`
aritméticos: `++, --, +, -, *, /, %`
comparación: `==, ¡=, <, <=, >, >=, !, &&, ||, ?:`
manejo bits: `&, |, ^, ~, <<, >>, >>>`
conversión: `(<tipo>)`

Conversión de tipos

Entre tipos compatibles se puede hacer asignación directa o utilizar un typecast.

```
byte b = 3;
int i = b; float f = i; //int a byte y float a int
b =(byte)i;  // hay que hacer un typecast
String s = Integer.toString(i);
b = Byte.parseByte(s);
```

Estructura condicional

`if`
`else`
`switch`
`case default`

```
if (<condición>) {
  <instrucciones>;
} else {
  <instrucciones>;
}
```

```
if (b != 0) {
  System.out.println(
    "x= "+a/b);
} else {
  System.out.println("Error");
}
```

```
switch (<expresión>) {
  case <valor>:
    <instrucciones>;
    [break;]
  case <valor>:
    <instrucciones>;
    [break;]
  ...
  [default:
    <instrucciones>;]
}
```

```
switch (opcion) {
  case 1:
    x = x * Math.sqrt(y);
    break;
  case 2:
  case 3:
    x = x / Math.log(y);
    break;
  default:
   System.out.println("Error");
}
```

Estructuras iterativas

`while`
`do`
`for`

```
while (<condición>) {
  <instrucciones>;
}
```

```
i = 0;
while (i < 10) {
  v[i]=0;
  i++;
}
```

```
do {
  <instrucciones>;
} while (<condición>)
```

```
i = 0;
do {
  v[i]=0;
  i++;
} while (i < 10)
```

```
for (<inicialización>;
    <comparación>; <incremento>) {
  <instrucciones>;
}
```

```
for (i = 0; i < 10; i++) {
  v[i]=0;
}
```

```
for (<tipo> <variable>
                  :<colección>) {
  <instrucciones>;
}          //(Java 5)
```

```
for (Complejo c: lista) {
   c.toString();
}
```

Sentencias de salto

`break`
`continue`
`return`
`exit`

`break;` fuerza la terminación inmediata de un bucle ó de un switch

`continue;` fuerza una nueva iteración del bucle o salta a una etiqueta.

`return [<valor>];` sale de la función, puede devolver un valor

`exit([int código]);` sale del programa, puede devolver un código

Clases y objetos

Clases

class

Cada clase ha de estar en un fichero separado con el mismo nombre de la clase y con extensión `.class`. Por convenio los identificadores de clase se escriben en mayúscula.

```
class <Clase> [extends <Clase_padre>][implement <interfaces>] {
  //declaración de atributos
  [visibilidad] [modificadores] <tipo> <atributo> [= valor];
  ...
  //declaración de constructor
  public <Clasee>(<argumentos>) {
    <instrucciones>;
  }
  //declaración de métodos
  [visibilidad] [modificadores] <tipo> <método>(<argumentos>) {
    <instrucciones>;
  }
  ...
}
```

donde: `[visibilidad] = public, protected o private`

`[modificadores] = final, static y abstract`

```
class Complejo {
  private double re, im;
  public Complejo(double re, double im) {
   this.re = re;  this.im = im;
  }
  public String toString() {
   return(new String(re + "+" + im + "i"));
  }
  public void suma(Complejo v) {
   re = re + v.re;
   im = im + v.im;
  }
}
```

Uso de objetos:

```
Complejo z, w;
z = new Complejo(-1.5, 3.0);
w = new Complejo(-1.2, 2.4);
z.suma(w);
System.out.println("Complejo: " + z.toString());
```

Sobrecarga

podemos escribir dos métodos con el mismo nombre si cambian sus parámetros.

```
public <tipo> <método>(<parámetros>) {
    <instrucciones>;
}
public <tipo> <método>(<otros parámetros>) {
    <otras instrucciones>;
}
```

ejemplo:

```
public Complejo sumar(Complejo c) {
  return new Complejo(re + c.re, im + c.im);
}
public Complejo sumar(double r, double i) {
```

```
    return new Complejo(re + r, im + i);
  }
```

Herencia:

extends
@Override
super.

```
class <Clase_hija> extends <Clase_padre> {
  ...
}
```

La clase hija va a heredar los atributos y métodos de la clase padre. Un objeto de la clase hija también lo es de la clase padre y de todos sus antecesores.

La clase hija puede volver a definir los métodos de la clase padre, en tal caso es recomendable (no obligatorio) indicarlo con `@Override`; de esta forma evitamos errores habituales cuando cambiamos algún carácter o parámetro. Si un método ha sido sobrescrito podemos acceder al de la clase padre con el siguiente prefijo **super.**`<método>(<parámetros>)`. Ver ejemplo del siguiente apartado.

Constructor

super()

Método que se ejecuta automáticamente cuando se instancia un objeto de una clase. Ha de tener el mismo nombre que la clase. Cuando se crea un objeto todos sus atributos se inicializan en memoria a cero y las referencias serán **null**. Una clase puede tener más de un constructor (véase sobrecarga). Un constructor suele comenzar llamando al constructor de la clase padre, para lo cual escribiremos como primera línea de código: **super**`(<parámetros>);`

Una clase puede tener más de un constructor (ver sobrecarga).

Un constructor suele comenzar llamando al constructor de la clase padre, para ello escribiremos como primera línea de código: **super**`(<parámetros>);`

```
class ComplejoAmpliado extends Complejo {
  private Boolean esReal;
  public ComplejoAmpliado(double re, double im) {
    super(re, im);
    esReal = im ==0;
  }
  public ComplejoAmpliado(double re) {
    super(re, 0);
    esReal = true;
  }
  @Override
  public Complejo sumar(double re, double im) {
    esReal = im == -this.im;
    return super.sumar(re,im);
  }
  public boolean esReal(){
    return esReal;
  }
}
```

Visibilidad

public
private
protected

La visibilidad indica quien puede acceder a un atributo o métodos. Se define antecediendo una de las palabras. (por defecto **public**)

public: accesibles por cualquier clase.

private: sólo son accesibles por la clase actual.

protected: sólo por la clase que los ha declarado y por sus descendientes.

`<si no indicamos nada>` sólo son accesibles por clases de nuestro paquete.

Modificadores

final
abstract
static

final: Se utiliza para declarar una constante (delante de un atributo), un método que no se podrá redefinir (delante de un método), o una clase de la que ya no se podrá heredar (delante de una clase).

abstract: Denota un método del cual no se escribirá código. Las clases con métodos abstractos no se pueden instanciar. Las clases descendientes deberán

escribir el código de sus métodos abstractos.

static: Se aplica a los atributos y métodos de una clase que pueden utilizarse sin crear un objeto que instancie dicha clase. El valor de un atributo estático, además, es compartido por todos los objetos de dicha clase.

Comparación y asignación de objetos

equals y ==

clone y =

Podemos comparar valores de variables con el operador ==, y asignar un valor a una variable con el operador =.

En cambio, el nombre de un objeto de una clase no contiene los valores de los atributos, sino la posición de memoria donde residen dichos valores de los atributos (referencia indirecta). Utilizaremos el operador == para saber si dos objetos ocupan la misma posición de memoria (son el mismo objeto), mientras que utilizaremos el método `equals(<Obeto>)` para saber si sus atributos tienen los mismos valores. Utilizaremos el operador = para asignar a un objeto otro objeto que ya existe (serán el mismo objeto) y **`clone()`** para crear una copia idéntica en un nuevo objeto.

Polimorfismo

instanceof

Se trata de declarar un objeto de una clase, pero instanciarlo como un descendiente de dicha clase (lo contrario no es posible):

```
<Clase_padre> <objeto> = new <Clase_hija>(<parámetros>);
```

Podemos preguntar al sistema si un objeto es de una determinada clase con:

```
<objeto> instanceof <Clase>
```

Podemos hacer un *tipecast* a un objeto para considerarlo de otra clase:

```
(<Clase>) <objeto>
```

Ejemplo:

```
Complejo c = new ComplejoAmpliado(12.4);
if (c instanceof Complejo)… //true
if (c instanceof ComplejoAmpliado)… //true
if (((ComplejoAmpliado)c).esReal())…
```

Recolector de basura

finalize()

Cuando termina el ámbito de un objeto (véase sección “Ámbito”) y no existen más referencias a él, el sistema lo elimina automáticamente.

```
{
  Complejo a;  // sólo a disponible, pero no inicializado
  {
    Complejo b = new Complejo(1.5,1.0); // Se crea un objeto
    a = b;  // Dos referencias a un mismo objeto
  }
  //  sólo a  disponible
} //  el objeto es destruido liberando su memoria
```

Para eliminar un objeto el sistema llama a su método `finalize()`. Podemos rescribir este método en nuestras clases:

```
@Override
protected void finalize() throws Throwable {
  try {
    close();        // cerramos fichero
  } finally {
    super.finalize();
  }
}
```

métodos con argumentos variables en número

```
[visibilidad] [modificadores] <tipo> <método>(<Clase>... args) {
  <instrucciones>;
}
```

Ejemplo:

```
public void sumar(Complejo... args) {
  for (int i = 0; i < args.length; i++) {
    re = re + args[i].re;
```

```
      im = im + args[i].im;
    }
  }
```

Interfaces

Interface

Clase completamente abstracta. No tiene atributos y ninguno de sus métodos tiene código. (En Java no existe la herencia múltiple, pero una clase puede implementar una o más interfaces, adquiriendo sus tipos).

```
interface <interface> [extends <interface_padre>] {
  [visibilidad] [modificadores] <tipo> <metodo1>(<argumentos>);
  [visibilidad] [modificadores] <tipo> <metodo2>(<argumentos>);
  ...
}

class <Nombre_clase> extends <clase_padre> implements
<interface1>, <interface2>, ... {
  ...
}
```

Otros

Paquetes

package

import

Los paquetes son una forma de organizar grupos de clases. Resuelven el problema del conflicto entre los nombres de las clases. Por ejemplo, la clase `Font` se ha creado en cientos de paquetes Java. Para referirnos a una de ellas es obligatorio indicar el paquete al que pertenece. Por ejemplo, `java.awt.Font`.

Al inicio del fichero de una clase se debe indicar su paquete:

```
package carpeta.subcarpeta.…;
```

Para usar una clase de otro paquete se indica su paquete:

```
java.awt.Font fuente=new java.awt.Font(…);
```

Para abreviar, tambien podemos importar la clase de un paquete:

```
import java.awt.Font;
```

y utilizar solo el nombre de la clase:

```
Font fuente=new Font(…);
```

Para importar todas las clases de un paquete:

```
import java.awt.*;
```

Excepciones

try

catch

finally

```
try {
  código donde se pueden producir excepciones
}
catch (TipoExcepcion1 NombreExcepcion) {
  código a ejecutar si se produce una excepción TipoExcepcion1
}
catch (TipoExcepcion2 NombreExcepcion) {
  código a ejecutar si se produce una excepción TipoExcepcion2
}
...
finally {
  código a ejecutar tanto si se produce una excepción como si no
}
```

Ejemplo:

```
String salario;
BufferedReader fichero1;
BufferedWriter fichero2;
try {
  fichero1 = new BufferedReader(new
```

```
                                FileReader("c:\\salarios.txt"));
    fichero2 = new BufferedWriter(new
                                FileWriter("c:\\salarios.new"));
    while ((salario = fichero1.readLine()) != null) {
      salario = (new Integer(Integer.parseInt(salario)*10)
                            .toString());
      fichero2.write(salario+"\n");
    }
  }
  catch (IOException e) {
    System.err.println(e);
  }
  catch (NumberFormatException e) {
    System.err.println("No es un número");
  }
  finally {
    fichero1.close(); fichero2.close();
  }
```

Hilos de ejecución

Thread

Creación de un nuevo hilo que llama una vez al método `hazTrabajo()`:

```
class MiHilo extends Thread {
  @Override public void run() {
    hazTrabajo();
  }
}
```

Para ejecutarlo:

```
MiHilo hilo = new MiHilo ();
hilo.start();
```

Creación de un nuevo hilo que llama continuamente al método `hazTrabajo()` y que puede ser pausado y detenido:

```
class MiHilo extends Thread {
  private boolean pausa,corriendo;
  public synchronized void pausar() {
    pausa = true;
  }
  public synchronized void reanudar() {
    pausa = false;
    notify();
  }
  public void detener() {
    corriendo = false;
    if (pausa) reanudar();
  }
  @Override public void run() {
    corriendo = true;
    while (corriendo) {
      hazTrabajo();
      synchronized (this) {
        while (pausa) {
          try {
            wait();
          } catch (Exception e) {}
        }
      }
    }
  }
}
```

Secciones críticas

Synchronized

Cada vez que un hilo de ejecución va a entrar en un método o bloque de instrucciones marcado con **synchronized** se comprueba si ya hay otro hilo dentro de la sección crítica de este objeto (formada por todos los bloques de instrucciones marcados con **synchronized**). Si ya hay otro hilo dentro, entonces el hilo actual es suspendido y ha de esperar hasta que la sección crítica quede libere. Para que un método pertenezca a la sección critica de objeto escribe:

```
public synchronized void metodo() {...}
```

o, para que un bloque de pertenezca a la sección critica de objeto escribe:

```
synchronized (this) {...}
```

Recuerda: La sección crítica se define a nivel de objeto no de clase. Solo se define una sección crítica por objeto.

Para conseguir una sección critica por clase escribe:

```
public static synchronized void metodo() {...}
```

o

```
synchronized (MiClase.class) {...}
```

Genericidad

Permite independizar el código del tipo de datos sobre el que se aplica.

```
Public class Caja<T> {
  private T dato;
  public void poner(T d) {dato = d;}
  public T sacar() {return dato;}
}
```

Inicio del Programa

main

```
class <Clase> {
  public static void main(String[] main) {
    <instrucciones>;
  }
}
```