Seminario 1

Introducción a Java

PROGRAMACIÓN 3

David Rizo, Pedro J. Ponce de León
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante

Contenidos

Contenidos

Características

Sintáxis básica

Programa principal

Compilación y ejecución

Tipos de datos escalares

Objetos

Excepciones

Cadenas

Arrays

Métodos

Escritura

Control de flujo

Paquetes

Librerías Java

CLASSPATH

Archivos JAR

Documentación

ANT

Características principales de Java

- Lenguaje orientado a objetos: (casi) todo son objetos
- Ficheros fuente: *.java*
- Se compila en *bytecode*: *.class*
- Librerías en ficheros *.jar*
- Lo ejecuta una *máquina virtual*: multiplataforma
- Entornos de desarrollo integrados (IDE) principales: *Netbeans*, *Eclipse*

David Rizo, Pedro J. Ponce de León

Sintaxis básica

Las reglas de nombrado de identificadores son básicamente las mismas que se usan para C++

```
// Este fichero se debe guardar en Clase.java
// Generalmente, cada clase se sitúa en un fichero
public class Clase {
  /* Todos los campos deben especificar la visibilidad */
  private int campo1;
  /**
   * Comentario Documentación
   */
  private float campo2; // los campos se inicializan a 0
  /* El constructor no devuelve nada */
  public Clase() {
     campo1 = 0;
  }
  /* Todos los métodos se definen inline */
  public int getCampo1() {
    return campo1;
  }
}
```

Constantes, estáticos

Constantes

Las constantes se definen usando la palabra reservada `final`

```
public final int KN=10;
```

Métodos y campos estáticos

Se definen usando la palabra reservada `static`

```
private static int contador=1;
public static final int KNN=10;
public static void incrementaContador () {
  contador++;
}
```

Programa principal

main

El punto de entrada a la aplicación *main* es un método constante estático

```
// esto es una clase normal
public class ClaseConMain {
  // que además tiene el método main
  public static final void main(String[] args) {
    // el array args contiene los argumentos
    // sin incluir (como en C++) el nombre del ejecutable
  }
}
```

Compilación y ejecución en línea de comando

Compilación

La compilación se realiza en el prompt

```
> javac ClaseConMain.java
```

Genera el fichero con *bytecode* ClaseConMain.class

Ejecución

que se ejecutará en prompt con la orden

```
> java ClaseConMain
```


David Rizo, Pedro J. Ponce de León

Tipos de datos básicos

Tipos escalares (no objetos)

Java es un lenguaje fuertemente tipado. Dispone de los tipos:

byte, short, int, long, float, double, char, boolean

Los literales se especifican:

```
float a = 10.3f;
double b = 10.3;
char c = 'a';
boolean d = true; // o false
```

Operadores

Disponemos de los mismos operadores que en C++

```
a++; if (a==1) b=2; a = (float)b;
```

Tipos escalares

Wrappers (objetos)

Cada tipo escalar tiene una clase equivalente:

`Byte`, `Integer`, `Float`, `Double`, `Char`, `Boolean`

que se inicializan

```
Integer a = null; // es nulo por defecto
a = new Integer(29);
int x = a.intValue(); // x será 29
```

Objetos

- Estos *wrappers* son objetos.
- Las variables que referencian objetos son realmente punteros y apuntan a `null` por defecto
- Hay que reservarles memoria con `new`
- No hay que liberarla explícitamente, lo hace el *Garbage Collector*

Objetos

Asignación

Al ser punteros la operación

```
Integer a = new Integer(10);
Integer b = a;
```

hace que b sea la misma instancia, la misma zona de memoria que b. Para duplicar habrá que crear un nuevo objeto con `new`

Object

La clase Object representa a *todos los objetos* de Java. Así, cualquier objeto de cualquier clase es también un objeto de la clase Object.

```
Object obj = new Integer(10); // Ok
obj = new Persona(); // Ok
```

Objetos

operador instanceof

La expresión

```
objeto  instanceof Clase
```

devuelve cierto si 'objeto' es un objeto de la clase 'Clase', y falso en caso contrario

Casting (conversión)

Es similar a C++:

```
int x = 10;
float f = (float) x;
```

Dado un objeto cualquiera, también podemos usar el operador de conversión para asignarlo a una referencia de tipo conocido:

```
Object cualquiera;
MiClase obj = (MiClase) cualquiera;
```

Nota: para hacer la conversión sin riesgo, debemos estar seguros de que 'cualquiera' es un objeto de tipo 'MiClase'.

Objetos

Comparación

La expresión

```
a==b
```

está comparando direcciones de memoria. Para comparar dos objetos debemos hacer:

```
a.equals(b)
```

El método 'equals'

Si queremos compara objetos de una clase creada por nosotros, debemos implementar el método 'equals'.

```
public boolean equals(Object obj)
```

El argumento de equals es una referencia a objeto de clase 'Object'. Esto implica que al método equals se le puede pasar un objeto de cualquier clase (aunque normalmente será uno del mismo tipo del objeto con que queremos compararlo).

David Rizo, Pedro J. Ponce de León

Implementación de 'equals'

Para implementar el método 'equals' hay que tener en cuenta que la operación de igualdad debe cumplir las propiedades reflexiva, simétrica y transitiva y asegurarnos de que

```
x.equals(null) == false // para cualquier x no nulo
```

Además, para poder comparar los atributos del argumento con los del objeto `this`, deberemos convertir el argumento a una referencia de nuestra clase. Por tanto, toda implementación del método equals debe realizar estas comprobaciones:

```
public boolean equals(Object obj) {
  if (obj == this) return true; // las dos referencias
      // apuntan al mismo objeto
  if (obj == null) return false;
  if (!(obj instanceof MiClase)) return false;
  MiClase elotro = (MiClase) obj;
  // a partir de aquí comparar los atributos de ambos
  // objetos ('this' y 'elotro') para determinar si éstos
  // son iguales o no.
  // ¡OJO! si los atributos son referencias a objetos,
  // hay que usar 'equals' para compararlos.
}
```

Objetos

Boxing

Cuando hacemos

```
Integer b = 3;
```

internamente se está haciendo

```
Integer b = new Integer(3);
```

Unboxing

y al contrario, al escribir

```
int x = new Integer(100);
```

internamente se está haciendo

```
int x = (new Integer(100)).intValue();
```

Excepciones

Concepto

- Una excepción es un mecanismo diseñado para manejar situaciones de error alterando el flujo normal de ejecución de un programa.
- Ejemplo de excepciones son el acceso a una dirección de memoria inválida, la división por cero, o la referencia a una posición negativa en un array.
- En su forma más básica, cuando se produce la excepción el método invocado aborta su ejecución y devuelve el control al método que lo invoca, operación que se repite hasta llegar al programa principal el cual para la ejecución de la aplicación.
- Las excepciones son objetos instancia de clases cuyo nombre suele tener la forma <Nombre>Exception.

Excepciones

Las dos excepciones con las que es más probable que nos encontremos son:

NullPointerException

Se lanza cuando estamos accediendo a una posición de memoria sin inicializar (para la que no se ha hecho un new). Por ejemplo:

```
Integer a, b;
if (a.equals(b)) {
// este if lanza la excepción NullPointerException
} .....
```

ArrayIndexOutOfBoundsException

Lanzada cuando se accede a una posición inválida de un array. Por ejemplo:

```
int [] v = new int[10];
v[20] = 3;
// esto lanza la excepción ArrayIndexOutOfBoundsException
```

Cadenas

String

Java dispone de una clase para trabajar con cadenas

```
String s = new String("Hola");
```

Recordar la comparación

```
s == "Hola" // mal
s.equals("Hola") // bien
```

toString()

Todas las clases suelen tener definido el método `toString()`.

```
Float f = new Float(20);
String s = f.toString();
```

Cadenas

Concatenación

Las cadenas se pueden concatenar con el operador +, si mezclamos otros tipos canónicos éstos se pasan a cadena

```
int i=100;
"El valor de i es = " + i;
```

Este código internamente crea 4 objetos, internamente hace

```
String s1 = new String("El valor de i es = ");
String s2 = new Integer(i).toString();
String s3 = s1.concatenate(s2); // que crea un objeto nuevo
```

StringBuilder

Para evitar la creación de tantos objetos podemos usar `StringBuilder` [1]

```
StringBuilder sb = new StringBuilder();
sb.append("El valor de i es = ");
sb.append(i);
sb.toString(); // objeto cadena
```

[1] `StringBuffer` para la versión sincronizada

Arrays

Los arrays se definen como los arrays dinámicos de C++

```
int [] v; // v es un puntero a null
```

que se incializa

```
v = new Integer[100];
```

Ahora los contenidos de `v`, es decir `v[0], v[1],` , etc... son `null`, se deben incializar

```
// v.length es la longitud reservada para el array
for (int i=0; i<v.length; i++) {
  v[i] = new Integer(0);
  // ó v[i] = 0 (equivalente por el boxing)
}
```

Se pueden crear literales array reservando también memoria

```
int [] v = new int []{1,2,3,4,5};
```

y se pueden copiar manualmente usando un bucle o con el método estático `arraycopy` de la clase `System`

```
int [] origen = new int []{1,2,3,4,5};
int [] destino = new int[origen.length];
System.arraycopy(origen, 0, destino, 0, origen.length);
```

Métodos, funciones

Métodos

Todo son objetos en Java: a las funciones miembro de un objeto se les llama métodos

Parámetros

Todos los parámetros se pasan por valor

```
void F(int a, String x, int [] v) {
  a=10; // este cambio no afectará al valor original
  x += "Hola"; // crea un nuevo objeto y no afecta al original
  v[2] = 7;
  // lo que se ha pasado por valor es
  // el puntero a v, v[2] sí se cambia en el original
}
```


David Rizo, Pedro J. Ponce de León

Salida por consola

Salida

Para imprimir por la salida estándard usaremos el método estático

```
System.out.print("Cadena"); // no imprime retorno de carro al final
System.out.println(10+3); // imprime retorno de carro al final
```

Para imprimir por la salida de error

```
System.err.println("Ha ocurrido un error...");
```

Control de flujo

En general no cambia respecto a C++. A partir de la versión 1.7 de Java existe la posibilidad de usar cadenas en los `switch`.

Bucles

Para recorrer vectores usaremos:

```
List<String> v = Arrays.asList("Azul", "Verde", "Rojo");

for (int i=0; i<v.size(); i++) {
  System.out.println(v.get(i));
}

for (String color: v) {
  System.out.println(color); // imprime un color por línea
}

// usando iteradores
Iterator<String> iterador = v.iterator();
while (iterador.hasNext()) {
  String color = iterador.next();
  System.out.println(color); // imprime un color por línea
}
```

Paquetes

package

Las clases se distribuyen físicamente en directorios. Éstos constituyen lo que se denomina `package`

Para que una clase esté en un paquete hay que:

- Guardar el fichero de la clase en el directorio del paquete
- Declarar al inicio del fichero el `package` al que pertenece, separando directorios (paquetes) con puntos

```
package prog3.ejemplos;
class Ejemplo {
}
```

El fichero `Ejemplo.java` se debe guardar en el directorio `prog3/ejemplos`.

David Rizo, Pedro J. Ponce de León

Paquetes

Modularización

No es obligatorio usar paquetes, pero es recomendable. Si queremos usar una clase de otro paquete debemos incluirla, tanto si es nuestra o de una librería

```
package prog3.ejemplos;
// clase de librería de Java
import java.util.ArrayList;

// clase nuestra de otro paquete
import prog3.otrosejemplos.Clase;

// esto incluye todas las clases de prog3.practicas.
// Por trazabilidad, es mejor no usar el *
import prog3.practicas.*;
```

Sólo se incluyen por defecto todas las clases de `java.lang` y por tanto no es necesario incluirlas explícitamente

```
// no es necesario, todos las clases de java.lang
// están incluidas por defecto
import java.lang.String;
```


David Rizo, Pedro J. Ponce de León

API Java

API

Java dispone de una extensa librería de clases que se puede consultar en `http://download.oracle.com/javase/6/docs/api/overview-summary.html`

Vectores

Como medio de almacenamiento lineal dinámico usaremos la clase `ArrayList`.

```
import java.util.ArrayList;
.....
ArrayList v = new ArrayList();
v.add(87); // esto internamente hace v.add(new Integer(87));
v.add(22); // hace más grande el vector

// get devuelve un Object,
// hay que hacer cast a Integer (que será 87)
Integer a = (Integer)v.get(0);
v.get(100); // lanza una excepción (error de ejecución)
```


David Rizo, Pedro J. Ponce de León

lsi

API Java

Clases genéricas

Podemos especificar el tipo almacenado en el vector, evitando tener que hacer casts

```
ArrayList<Integer> v = new ArrayList<Integer>();
v.add(87); // esto internamente hace v.add(new Integer(87));
Integer a = v.get(0); // no necesitamos hacer cast
System.out.println(v.size()); // size() devuelve el tamaño
```

Inicialización

Podemos inicializar cómodamente los vectores con

```
List<String> v = Arrays.asList("Azul", "Verde", "Rojo");
// v es inicializado como un objeto ArrayList
```

CLASSPATH

ClassNotFoundException

Esta excepción aparece normalmente al iniciar un programa Java. Antes de comenzar a ejecutar el programa principal, la máquina virtual debe cargar todos los archivos .class necesarios. Si no encuentra alguno, lanza esta excepción.

Ejemplo

```
mihome> java Main
Exception in thread "main" java.lang.NoClassDefFoundError: Main
Caused by: java.lang.ClassNotFoundException: Main
at java.net.URLClassLoader\$1.run(URLClassLoader.java:202)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
at sun.misc.Launcher\$AppClassLoader.loadClass(Launcher.java:301)
at java.lang.ClassLoader.loadClass(ClassLoader.java:247)
```

Pero ¿dónde deben estar esos archivos? En el *classpath*.

CLASSPATH

classpath

El *classpath* es la lista de directorios donde Java busca los archivos .class necesarios para ejecutar una aplicación.
Por defecto,

- el directorio actual
- librerías del JRE (Java Runtime Environment), donde se encuentran los archivos .class de la API de Java.

CLASSPATH

Supongamos que nuestro programa principal está compilado en un archivo llamado `Main.class` que reside en `/home/mihome/miapp`.

Caso 1

Todas nuestras clases están en un mismo directorio. No usamos `package`. Desde ese directorio,

```
/home/mihome/miapp> java Main
```

CLASSPATH

Caso 2

Ejecutamos `java` desde un directorio distinto al que contiene nuestros `.class`. Hay que definir el *classpath*:

Opcion 1

Definir la variable de entorno **CLASSPATH** con la lista de directorios donde están los .class (mejor usar rutas absolutas)

```
.../otrodirectorio> export CLASSPATH=/home/mihome/miapp

.../otrodirectorio> java Main
```

Opcion 2

Usar la opcion **-cp** o **-classpath** de `java`:

```
.../otrodirectorio> java -cp /home/mihome/miapp Main
```

CLASSPATH

Caso 3

Los .class están repartidos en varios directorios.

```
> export CLASSPATH=/home/mihome/milibjava:/home/mihome/miapp
> java Main
```

o bien usar **-cp**. OJO: la opcion '-cp' anula a CLASSPATH. Se debe usar una u otra, pero no ambas a la vez.

Paquetes y *classpath*

Cuando nuestras clases están organizadas en paquetes.
Supongamos que tenemos lo siguiente:

Estructura del proyecto

modelo/MiClase.java:

```
package modelo;
public class MiClase {...}
```

mains/Main.java:

```
package mains;
public class Main {...}
```

modelo/m2/OtraClase.java:

```
package modelo.m2;
public class OtraClase {...}
```

classpath deberá contener **el directorio padre** de la estructura de paquetes.

Paquetes y *classpath*

Supongamos que ese directorio del proyecto es `/home/mihome/miapp`. Si quiero usar **OtraClase** en Main.java:

```
import modelo.m2.OtraClase;
```

Al ejecutar

```
.../otrodir>java -cp /home/mihome/miapp Main
```

para poder ejecutar la clase Main, 'java' buscará en los directorios del *classpath* un directorio `modelo/m2` y dentro de éste el archivo `OtraClase.class`.

Archivos JAR

jar es una utilidad de Java (similar a **tar**) para empaquetar en un único fichero con extensión **.jar** una estructura de directorios. Se suele usar para archivos .class.

JAR

Para empaquetar, desde el directorio de trabajo:

```
> jar cvf MisClases.jar *.class
```

Ahora podemos llevarnos MisClases.jar donde queramos (p. ej. /home/mihome/libs) y, desde cualquier lugar:

```
> java -cp /home/mihome/libs/MisClases.jar Main
```

Para ver el contenido de un archivo .jar:

```
> jar tvf MisClases.jar
```

Documentación I

Javadoc

En java se utiliza un formato basado en anotaciones embebido en comentarios. Éstos se inician con `/**` y los tipos de anotaciones comienzan por `@`:

```
package paquete;
/**
 * Clase de ejemplo: documentamos brevemente el cometido
 * de la clase
 * @author drizo
 * @version 1.8.2011
 */
 public class Ejemplo {

 /**
  * Esto es un campo que vale para ...
  */
  private int x;

  private int y; // esto no sale en el javadoc
```

Documentación II

```
  /**
   * Constructor: hace esta operación....
   * @param ax Es el radio de ...
    *@param ab Si es cierto pasa ...
    */
    public Ejemplo(int ax, boolean ab) {
      ....
    }

   /**
    * Getter.
    @return x: sabemos que siempre es mayor que cero...
    */
    public double getX() {
        return x;
    }
}
```

Documentación III

Generación

La documentación en html se genera mediante la orden

```
javadoc -d doc paquete otropaquete
```

genera un directorio `doc` con la documentación de las clases en los paquetes `paquete` y `otropaquete`.

ANT

ant

ant es una herramienta para automatizar las diversas tareas relativas a la compilacion, generación de documentación, archivos jar, etc. Es similar a 'make'. En *Programación 3* lo usaremos como parte del script de corrección de prácticas.

Tutorial de 'ant'

En el enlace siguiente tienes un breve tutorial en castellano:
`http://www.chuidiang.com/java/herramientas/ant.php`

David Rizo, Pedro J. Ponce de León

lsi