

# Android y Java para Dispositivos Móviles

## Sesión 1: Introducción a Java



# Puntos a tratar

- Introducción a Java
- Conceptos de POO
- Herencia e interfaces
- Colecciones de datos
- Excepciones
- Entrada y salida (ficheros, recursos y red)
- Serialización de datos
- Depuración



# Java

- *Java* es un lenguaje OO creado por *Sun Microsystems* para poder funcionar en distintos tipos de procesadores y máquinas.
- Similar a C o C++, pero con algunas características propias (gestión de hilos, ejecución remota, etc)
- Independiente de la plataforma, gracias a la JVM (*Java Virtual Machine*), que interpreta los ficheros objeto
- Se dispone de antemano de la API (*Application Programming Interface*) de clases de Java.



# Clases

- *Clases*: con la palabra *class* y el nombre de la clase

```
class MiClase  
{  
    ...  
}
```

- Como nombre utilizaremos un sustantivo
- Puede estar formado por varias palabras
- Cada palabra comenzará con mayúscula, el resto se dejará en minúscula
  - Por ejemplo: `DataStream`
- Si la clase contiene un conjunto de métodos estáticos o constantes relacionadas pondremos el nombre en plural
  - Por ejemplo: `Resources`



# Campos y variables

- *Campos y variables*: simples o complejos
- Utilizaremos sustantivos como nombres

```
Properties propiedades;  
File ficheroEntrada;  
int numVidas;
```

- Puede estar formado por varias palabras, con la primera en minúsculas y el resto comenzando por mayúsculas y el resto en minúsculas
  - Por ejemplo: `numVidas`
- En caso de tratarse de una colección de elementos, utilizaremos plural
  - Por ejemplo: `clientes`
- Para variables temporales podemos utilizar nombres cortos, como las iniciales de la clase a la que pertenezca, o un carácter correspondiente al tipo de dato

```
int i;  
Vector v;  
DataInputStream dis;
```



# Constantes

- *Constantes*: Se declaran como *final* y *static*

```
final static String TITULO_MENU = "Menu";  
final static int ANCHO_VENTANA = 640;  
final static double PI = 3.1416;
```

- El nombre puede contener varias palabras
- Las palabras se separan con '\_'
- Todo el nombre estará en mayúsculas
  - Por ejemplo: MAX\_MENSAJES



# Métodos

- *Métodos*: con el tipo devuelto, nombre y parámetros

```
void imprimir(String mensaje)
{
    ...// Código del método
}
Vector insertarVector(Object elemento, int posicion)
{
    ...// Código del método
}
```

- Los nombres de los métodos serán verbo
- Puede estar formados por varias palabras, con la primera en minúsculas y el resto comenzando por mayúsculas y el resto en minúsculas
  - Por ejemplo: `imprimirDatos`



# Constructores

- *Constructores*: se llaman igual que la clase, y se ejecutan con el operador *new* para reservar memoria

```
MiClase()  
{  
    ...// Codigo del constructor  
}  
MiClase(int valorA, Vector valorV)  
{  
    ...// Codigo del otro constructor  
}
```

- No hace falta destructor, de eso se encarga el *garbage collector*
- Constructor superclase: `super (...)`





# Paquetes

- *Paquetes*: organizan las clases en una jerarquía de paquetes y subpaquetes
- Para indicar que una clase pertenece a un paquete o subpaquete se utiliza la palabra *package* al principio de la clase

```
package paquete1.subpaquete1;  
class MiClase {
```

- Para utilizar clases de un paquete en otro, se colocan al principio sentencias *import* con los paquetes necesarios:

```
package otropaquete;  
import paquete1.subpaquete1.MiClase;  
import java.util.*;  
class MiOtraClase {
```



# Paquetes

- Si no utilizamos sentencias *import*, deberemos escribir el nombre completo de cada clase del paquete no importado (incluyendo subpaquetes)

```
class MiOtraClase {  
    paquete1.subpaquete1.MiClase a = ...; // Sin import  
    MiClase a = ...;                       // Con import
```

- Los paquetes se estructuran en directorios en el disco duro, siguiendo la misma jerarquía de paquetes y subpaquetes

```
./paquete1/subpaquete1/MiClase.java
```



# Paquetes

- Siempre se deben incluir las clases creadas en un paquete
  - Si no se especifica un nombre de paquete la clase pertenecerá a un paquete “sin nombre”
  - No podemos importar clases de paquetes “sin nombre”, las clases creadas de esta forma no serán accesibles desde otros paquetes
  - Sólo utilizaremos paquetes “sin nombre” para hacer una prueba rápida, nunca en otro caso



# Convenciones de paquetes

- El nombre de un paquete deberá constar de una serie de palabras simples siempre en minúsculas
  - Se recomienda usar el nombre de nuestra DNS al revés  
`jtech.ua.es` → `es.ua.jtech.prueba`
- Colocar las clases interdependientes, o que suelen usarse juntas, en un mismo paquete
- Separar clases volátiles y estables en paquetes diferentes
- Hacer que un paquete sólo dependa de paquetes más estables que él
- Si creamos una nueva versión de un paquete, daremos el mismo nombre a la nueva versión sólo si es compatible con la anterior



# Tipo enumerado

```
enum EstadoCivil {soltero, casado, divorciado};
```

```
EstadoCivil ec = EstadoCivil.casado;  
ec = EstadoCivil.soltero;
```

```
switch(ec) {  
    case soltero:  
        System.out.println("Es soltero"); break;  
    case casado:  
        System.out.println("Es casado"); break;  
    case divorciado:  
        System.out.println("Es divorciado"); break;  
}
```



# Otras características

- Imports estáticos

```
import static java.lang.Math;  
...  
double raiz = sqrt(1252.2);
```

- Argumentos variables

```
public void miFunc(String param, int... args) {  
    for(int i: args) { ... }  
}
```

- Anotaciones (metainformación)
  - P.ej., @deprecated



# Convenciones generales

- Indentar el código uniformemente
- Limitar la anchura de las líneas de código (para impresión)
- Utilizar líneas en blanco para separar bloques de código
- Utilizar espacios para separar ciertos elementos en una línea
- Documentación:
  - Utilizar `/* . . . */` para esconder código sin borrarlo
  - Utilizar `// . . .` para detalles de la implementación
  - Utilizar javadoc para describir la interfaz de programación



# Modificadores de acceso

- Las clases y sus elementos admiten unos modificadores de acceso:
  - *privado*: el elemento es accesible sólo desde la clase en que se encuentra
  - *protegido*: el elemento es accesible desde la propia clase, desde sus subclases, y desde clases del mismo paquete
  - *público*: el elemento es accesible desde cualquier clase
  - *paquete*: el elemento es accesible desde la propia clase, o desde clases del mismo paquete.





# Modificadores de acceso

- *private* se utiliza para elementos PRIVADOS
- *protected* se utiliza para elementos PROTEGIDOS
- *public* se utiliza para elementos PUBLICOS
- No se especifica nada para elementos PAQUETE

```
public class MiClase {  
    private int n;  
    protected void metodo() { ... }
```

- Todo fichero Java debe tener una y solo una clase pública, llamada igual que el fichero (más otras clases internas que pueda tener)



# Otros modificadores

- *abstract*: para definir clases y métodos abstractos
- *static*: para definir elementos compartidos por todos los objetos que se creen de la misma clase
  - NOTA: dentro de un método estático sólo podemos utilizar elementos estáticos, o elementos que hayamos creado dentro del propio método
- *final*: para definir elementos no modificables ni heredables
- *synchronized*: para elementos a los que no se puede acceder al mismo tiempo desde distintos hilos de ejecución

```
public abstract class MiClase {  
    public static final int n = 20;  
    public abstract void metodo();  
    ...  
}
```



# Herencia y polimorfismo

- **Herencia:** definir una clase a partir de otra existente
  - La nueva clase “hereda” todos los campos y métodos de la clase a partir de la que se crea, y aparte puede tener los suyos propios
  - *Ejemplo:* a partir de una clase *Animal* podemos definir otras más concretas como *Pato*, *Elefante*...
- **Polimorfismo:** si tenemos un método en cualquier clase que sea *dibuja (Animal a)*, podemos pasarle como parámetro tanto un objeto *Animal* como cualquier subtipo que herede directa o indirectamente de él (*Elefante*, *Pato*...)



# Clases abstractas e interfaces

- Una *clase abstracta* es una clase que deja algunos métodos sin código, para que los rellenen las subclases que hereden de ella

```
public abstract class MiClase {  
    public abstract void metodo1();  
    public void metodo2() {  
        ...  
    }  
}
```

- Un *interfaz* es un elemento que sólo define la cabecera de sus métodos, para que las clases que implementen dicha interfaz rellenen el código según sus necesidades.

```
public interface Runnable {  
    public void run();  
}
```

- Asignaremos un nombre a los interfaces de forma similar a las clases, pudiendo ser en este caso adjetivos o sustantivos.



# Herencia e interfaces

- Herencia

- Definimos una clase a partir de otra que ya existe
- Utilizamos la palabra *extends* para decir que una clase hereda de otra (Pato *hereda de* Animal):

```
class Pato extends Animal
```

- Relación “es”: Un pato *ES* un animal

- Interfaces

- Utilizamos la palabra *implements* para decir que una clase implementa los métodos de una interfaz

```
class MiHilo implements Runnable {  
    public void run() {  
        ... // Código del método  
    }  
}
```

- Relación “actúa como”: MiHilo *ACTÚA COMO* ejecutable



# Polimorfismo

- Si una variable es del tipo de la superclase, podemos asignarle también un objeto de la clase hija

```
Animal a = new Pato();
```

- Si una variable es del tipo de una interfaz implementada por nuestra clase, podemos asignarle también un objeto de esta clase

```
Runnable r = new MiHilo();
```

- Sólo se puede heredar de una clase, pero se pueden implementar múltiples interfaces:

```
class Pato extends Animal implements Runnable, ActionListener
```



# Punteros *this* y *super*

- *this* se utiliza para hacer referencia a los elementos de la propia clase:

```
class MiClase {  
    int i;  
    MiClase(int i) {  
        this.i = i;    // i de la clase = i del parámetro  
    }  
}
```

- *super* se utiliza para llamar al mismo método en la superclase:

```
class MiClase extends OtraClase{  
    MiClase(int i) {  
        super(i);    // Constructor de OtraClase(...)  
    }  
}
```



# Object

- Clase base de todas las demás
  - Todas las clases heredan en última instancia de ella
- Es importante saber las dependencias (herencias, interfaces, etc) de una clase para saber las diferentes formas de instanciarla o referenciarla (polimorfismo)





# Ejemplo de polimorfismo

- Por ejemplo, si tenemos:

```
public class MiClase extends Thread  
                        implements List
```

- Podremos referenciar un objeto *MiClase* de estas formas:

```
MiClase mc = new MiClase();  
Thread t = new MiClase();  
List l = new MiClase();  
Object o = new MiClase();
```



# Object: objetos diferentes

- También es importante distinguir entre entidades independientes y referencias:

```
MiClase mc1 = new MiClase();  
MiClase mc2 = mc1;  
// Es distinto a:  
MiClase mc2 = (MiClase)(mc1.clone());
```

- El método *clone* de cada objeto sirve para obtener una copia en memoria de un objeto con los mismos datos, pero con su propio espacio
  - No realiza una copia en profundidad
  - Si queremos hacer copias de los objetos que tenga como campos debe sobrescribir este método



# Object: comparar objetos

- Cuando queremos comparar dos objetos entre sí (por ejemplo, de la clase *MiClase*), no se hace así: `if (mc1 == mc2)`
- Sino con su método *equals*: `if (mc1.equals(mc2))`
- Deberemos redefinir este método en las clases donde lo vayamos a usar, para asegurarnos de que los objetos se comparen bien
  - Notar que la clase *String*, es un subtipo de *Object* por lo que para comparar cadenas...:

```
if (cadena == "Hola") ... // NO
if (cadena.equals("Hola")) ... // SI
```



# Object: representar en cadenas

- Muchas veces queremos imprimir un objeto como cadena. Por ejemplo, si es un punto geométrico, sacar su coordenada X, una coma, y su coordenada Y
- La clase *Object* proporciona un método *toString* para definir cómo queremos que se imprima un objeto. Podremos redefinirlo a nuestro gusto

```
public class Punto2D {  
    ...  
    public String toString()  
    {  
        System.out.println("(" + x + ", " + y + ")");  
    }  
}  
...  
Punto2D p = ...;  
System.out.println(p); // Sacará (x, y) del punto
```



# Otras clases

- La clase *Math* proporciona una serie de métodos (estáticos) útiles para diferentes operaciones matemáticas (logaritmos, potencias, exponenciales, máximos, mínimos, etc)
- Otras clases útiles son la clase *Calendar* (para trabajar con fechas y horas), la clase *Currency* (para monedas), y la clase *Locale* (para situarnos en las características de fecha, hora y moneda de una región del mundo)



# Getters y Setters

- Es buena práctica de programación declarar como privados todos los campos de las clases
- Para acceder a ellos utilizaremos métodos
  - *Getters* para obtener el valor del campo
  - *Setters* para modificar el valor del campo
- Estos métodos tendrán prefijo `get` y `set` respectivamente, seguido del nombre del campo al que acceden, pero comenzando por mayúscula
  - Por ejemplo: `getLogin()`, `setLogin(String login)`
- El *getter* para campos booleanos tendrá prefijo `is` en lugar de `get`
  - Por ejemplo: `isAdministrador()`



# Colecciones

- En el paquete `java.util`
- Representan grupos de objetos, llamados elementos
- Podemos encontrar de distintos tipos, según si sus elementos están ordenados, si permiten repetir elementos, etc
- La interfaz `Collection` define el esqueleto que deben tener todos los tipos de colecciones
- Por tanto, todos tendrán métodos generales como:
  - `boolean add(Object o)`
  - `boolean remove(Object o)`
  - `boolean contains(Object o)`
  - `void clear()`
  - `boolean isEmpty()`
  - `Iterator iterator()`
  - `int size()`
  - `Object[] toArray()`



# Listas de elementos

- La interfaz `List` hereda de `Collection`
  - Operaciones propias de una colección tipo *lista*
  - Los elementos tienen un orden (posición en la lista)
- Así, tendremos otros nuevos métodos, además de los de `Collection`:
  - `void add(int posicion, Object o)`
  - `Object get(int indice)`
  - `int indexOf(Object o)`
  - `Object remove(int indice)`
  - `Object set(int indice, Object o)`





# Tipos de listas

- ArrayList: implementa una lista de elementos mediante un *array* de tamaño variable
  - *NO sincronizado*
- Vector: existe desde las primeras versiones de Java, después se acomodó al marco de colecciones implementando la interfaz List.
  - *Similar a ArrayList, pero SINCRONIZADO. Tiene métodos anteriores a la interfaz List:*
    - `void addElement(Object o) / boolean removeElement(Object o)`
    - `void insertElementAt(Object o, int posicion)`
    - `void removeElementAt(Object o, int posicion)`
    - `Object elementAt(int posicion)`
    - `void setElementAt(Object o, int posicion)`
    - `int size()`
- LinkedList: lista doblemente enlazada. Útil para simular pilas o colas
  - `void addFirst(Object o) / void addLast(Object o)`
  - `Object getFirst() / Object getLast()`
  - `Object removeFirst() / Object removeLast()`



# Mapas

- No forman parte del marco de colecciones
- Se definen en la interfaz Map, y sirven para relacionar un conjunto de claves (*keys*) con sus respectivos valores
- Tanto la clave como el valor pueden ser cualquier objeto
  - `Object get(Object clave)`
  - `Object put(Object clave, Object valor)`
  - `Object remove(Object clave)`
  - `Set keySet()`
  - `int size()`



# Tipos de mapas

- **HashMap:** Utiliza una tabla *hash* para almacenar los pares *clave=valor*.
  - Las operaciones básicas (get y put) se harán en tiempo constante si la dispersión es adecuada
  - La iteración es más costosa, y el orden puede diferir del orden de inserción
- **Hashtable:** como la anterior, pero **SINCRONIZADA**. Como Vector, está desde las primeras versiones de Java
  - `Enumeration keys()`
- **TreeMap:** utiliza un árbol para implementar el mapa
  - El coste de las operaciones es logarítmico
  - Los elementos están ordenados ascendentemente por clave



# Genéricos

- Colecciones de tipos concretos de datos
  - A partir de JDK 1.5
  - Aseguran que se utiliza el tipo de datos correcto

```
ArrayList<String> a = new  
    ArrayList<String>();  
a.add("Hola");  
String s = a.get(0);
```

- Podemos utilizar genéricos en nuestras propias clases



# Enumeraciones e iteradores

- Las enumeraciones y los iteradores no son tipos de datos en sí, sino objetos útiles a la hora de recorrer diferentes tipos de colecciones
- Con las *enumeraciones* podremos recorrer secuencialmente los elementos de una colección, para sacar sus valores, modificarlos, etc
- Con los *iteradores* podremos, además de lo anterior, eliminar elementos de una colección, con los métodos que proporciona para ello.



# Enumeraciones

- La interfaz `Enumeration` permite consultar secuencialmente los elementos de una colección
- Para recorrer secuencialmente los elementos de la colección utilizaremos su método `nextElement`:

```
Object item = enum.nextElement();
```

- Para comprobar si quedan más elementos que recorrer, utilizamos el método `hasMoreElements`:

```
if (enum.hasMoreElements()) ...
```

# Enumeraciones

- Con lo anterior, un bucle completo típico para recorrer una colección utilizando su enumeración de elementos sería:

```
// Obtener la enumeracion
Enumeration enum = coleccion.elements();
while (enum.hasMoreElements())
{
    Object item = enum.nextElement();
    ...// Convertir item al objeto adecuado y
        // hacer con el lo que convenga
}
```



# Iteradores

- La interfaz `Iterator` permite iterar secuencialmente sobre los elementos de una colección
- Para recorrer secuencialmente los elementos de la colección utilizaremos su método `next`:

```
Object item = iter.next();
```

- Para comprobar si quedan más elementos que recorrer, utilizamos el método `hasNext`:

```
if (iter.hasNext()) ...
```

- Para eliminar el elemento de la posición actual del iterador, utilizamos su método `remove`:

```
iter.remove();
```



# Iteradores

- Con lo anterior, un bucle completo típico para recorrer una colección utilizando su iterador sería:

```
// Obtener el iterador
Iterator iter = coleccion.iterator();
while (iter.hasNext())
{
    Object item = iter.next();
    ...// Convertir item al objeto adecuado y
        // hacer con el lo que convenga, por ejemplo
    iter.remove();
}
```



# Bucles sin iteradores

- Nueva versión del for en JDK 1.5
- Permite recorrer tanto *arrays* como colecciones
- Previene salirse del rango de forma segura

```
List<String> lista = obtenerLista();  
for(String cadena: lista)  
    System.out.println (cadena);
```



# Polimorfismo e interfaces

- Hacer referencia siempre mediante la interfaz
  - Permite cambiar la implementación sin afectar al resto del programa

```
public class Cliente {  
    List<Cuenta> cuentas;  
    public Cliente() {  
        this.cuentas = new ArrayList<Cuenta>();  
    }  
    public List<Cuenta> getCuentas() {  
        return cuentas;  
    }  
}
```



# Wrappers

- Los tipos simples (`int`, `char`, `float`, `double`, etc) no pueden incluirse directamente en colecciones, ya que éstas esperan subtipos de `Object` en sus métodos
- Para poderlos incluir, se tienen unas clases auxiliares, llamadas *wrappers*, para cada tipo básico, que lo convierten en objeto complejo
- Estas clases son, respectivamente, `Integer`, `Character`, `Float`, `Double`, etc.
- Encapsulan al tipo simple y ofrecen métodos útiles para poder trabajar con ellos

# Wrappers

- Si quisiéramos incluir un entero en un `ArrayList`, lo podríamos hacer así:

```
int a;  
ArrayList al = new ArrayList();  
al.add(new Integer(a));
```

- Si quisiéramos recuperar un entero de un `ArrayList`, lo podríamos hacer así:

```
Integer entero = (Integer)(al.get(posicion));  
int a = entero.intValue();
```



# Autoboxing

- Nueva característica de JDK 1.5
- Conversiones automáticas entre tipos básicos y sus *wrappers*

```
Integer n = 10;  
int num = n;
```

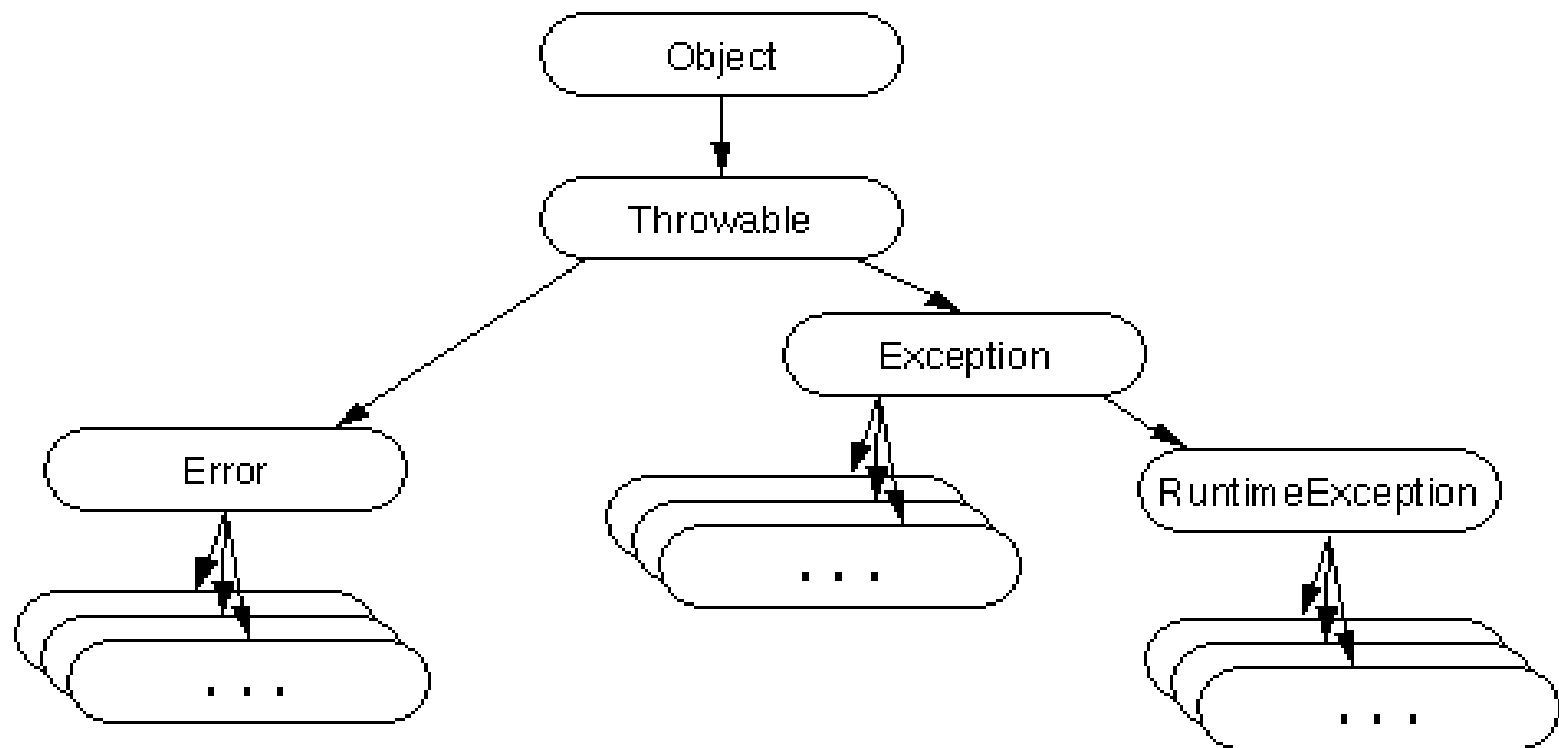
```
List<Integer> lista= new ArrayList<Integer>();  
lista.add(10);  
int elem = lista.get(0);
```



# Excepciones

- Excepción: Evento que sucede durante la ejecución del programa y que hace que éste salga de su flujo normal de ejecución
  - Se lanzan cuando sucede un error
  - Se pueden capturar para tratar el error
- Son una forma elegante para tratar los errores en Java
  - Separa el código normal del programa del código para tratar errores.

# Jerarquía







# Tipos de Excepciones

- **Checked: Derivadas de Exception**
  - Es obligatorio capturarlas o declarar que pueden ser lanzadas
  - Se utilizan normalmente para errores que pueden ocurrir durante la ejecución de un programa, normalmente debidos a factores externos
  - P.ej. Formato de fichero incorrecto, error leyendo disco, etc
- **Unchecked: Derivadas de RuntimeException**
  - Excepciones que pueden ocurrir en cualquier fragmento de código
  - No hace falta capturarlas (es opcional)
  - Se utilizan normalmente para errores graves en la lógica de un programa, que no deberían ocurrir
  - P.ej. Puntero a null, fuera de los límites de un array, etc



# Creación de excepciones

- Podemos crear cualquier nueva excepción creando una clase que herede de `Exception` (checked), `RuntimeException` (unchecked) o de cualquier subclase de las anteriores.

```
public class MiExcepcion extends Exception {  
    public MiExcepcion (String mensaje) {  
        super(mensaje);  
    }  
}
```



# Try – catch – finally

```
try {  
    // Código regular del programa  
    // Puede producir excepciones  
} catch (TipoDeExcepcion1 e1) {  
    // Código que trata las excepciones de tipo  
    // TipoDeExcepcion1 o subclases de ella.  
    // Los datos sobre la excepción los encontraremos  
    // en el objeto e1.  
    ...  
} catch (TipoDeExcepcionN eN) {  
    // Código que trata las excepciones de tipo  
    // TipoDeExcepcionN o subclases de ella.  
} finally {  
    // Código de finalización (opcional)  
}
```



# Ejemplos

- Sólo captura `ArrayOutOfBoundsException`

```
int [] hist = leeHistograma();
try {
    for(int i=1;;i++) hist[i] += hist[i-1];
} catch(ArrayOutOfBoundsException e) {
    System.out.println("Error: " + e.getMessage());
}
```

- Captura cualquier excepción

```
int [] hist = leeHistograma();
try {
    for(int i=1;;i++) hist[i] += hist[i-1];
} catch(Exception e) {
    System.out.println("Error: " + e.getMessage());
}
```



# Lanzar una excepción

- Si la excepción es *checked*, declarar que el método puede lanzarla con `throws`

```
public void leeFichero() throws ParseException
{
    ...
    throw new ParseException(mensaje, linea);
    ...
}
```



# Capturar o propagar

- Si un método lanza una excepción *checked* deberemos
  - Declarar que puede ser lanzada para propagarla al método llamador

```
public void init() throws ParseException {  
    leeFichero();  
}
```

- O capturarla para que deje de propagarse

```
try {  
    leeFichero();  
} catch (ParseException e) {  
    System.out.println("Error en línea " +  
        e.getOffset() + ": " + e.getMessage());  
}
```

- Si es unchecked
  - Se propaga al método llamante sin declarar que puede ser lanzada
  - Parará de propagarse cuando sea capturada
  - Si ningún método la captura, la aplicación terminará automáticamente mostrándose la traza del error producido



# Serialización. Flujos de E/S

- Las aplicaciones muchas veces necesitan enviar datos a un determinado destino o leerlos de una determinada fuente
  - Ficheros en disco, red, memoria, otras aplicaciones, etc
  - Esto es lo que se conoce como E/S
- Esta E/S en Java se hace mediante flujos (*streams*)
  - Los datos se envían en serie a través del flujo
  - Se puede trabajar de la misma forma con todos los flujos, independientemente de su fuente o destino

Todos derivan de las mismas clases



# Tipos de flujos según el tipo de datos

- Según el tipo de datos que transportan, distinguimos
  - Flujos de bytes (con sufijos `InputStream` y `OutputStream`)
  - Flujos de caracteres (con sufijos `Reader` y `Writer`)
- Superclases

	Entrada	Salida
Bytes	<code>InputStream</code>	<code>OutputStream</code>
Caracteres	<code>Reader</code>	<code>Writer</code>





# Tipos de flujos según su propósito

- Distinguimos:

- Canales de datos

Simplemente llevan datos de una fuente a un destino

Ficheros: `FileInputStream`, `FileReader`,  
`FileOutputStream`, `FileWriter`

Memoria: `ByteArrayInputStream`, `CharArrayReader`, ...

Tuberías: `PipedInputStream`, `PipedReader`, `PipedWriter`,  
...

- Flujos de procesamiento

Realizan algún procesamiento con los datos

Impresión: `PrintWriter`, `PrintStream`

Conversores de datos: `DataOutputStream`, `DataInputStream`

Bufferes: `BufferedReader`, `BufferedInputStream`, ...



# Acceso a los flujos

- Todos los flujos tienen una serie de métodos básicos

Flujos	Métodos
InputStream, Reader	read, reset, close
OutputStream, Writer	write, flush, close

- Los flujos de procesamiento
  - Se construyen a partir de flujos canales de datos
  - Los extienden proporcionando métodos de más alto nivel, p.ej:

Flujos	Métodos
BufferedReader	readLine
DataOutputStream	writeInt, writeUTF, ...
PrintStream, PrintWriter	print, println



# Objetos de la E/S estándar

- En Java también podemos acceder a la entrada, salida y salida de error estándar
- Accedemos a esta E/S mediante flujos
- Estos flujos se encuentran como propiedades estáticas de la clase `System`

	Tipo de flujo	Propiedad
Entrada	<code>InputStream</code>	<code>System.in</code>
Salida	<code>PrintStream</code>	<code>System.out</code>
Salida de error	<code>PrintStream</code>	<code>System.err</code>



# Salida estándar

- La salida estándar se ofrece como flujo de procesamiento `PrintStream`
  - Con un `OutputStream` a bajo nivel sería demasiado incómoda la escritura
- Este flujo ofrece los métodos `print` y `println` que permiten imprimir cualquier tipo de datos básico

- En la salida estándar

```
System.out.println("Hola mundo");
```

- En la salida de error

```
System.err.println("Error");
```



# Flujos de ficheros

- Canales de datos para acceder a ficheros

	Entrada	Salida
Caracteres	FileReader	FileWriter
Binarios	FileInputStream	FileOutputStream

- Se puede acceder a bajo nivel directamente de la misma forma que para cualquier flujo
- Podemos construir sobre ellos flujos de procesamiento para facilitar el acceso de estos flujos



# Lectura y escritura de ficheros

```
public void copia_fichero() {  
    int c;  
    try {  
        FileReader in = new FileReader("fuente.txt");  
        FileWriter out = new FileWriter("destino.txt");  
        while( (c = in.read()) != -1)  
        {  
            out.write(c);  
        }  
        in.close();  
        out.close();  
    } catch(FileNotFoundException e1) {  
        System.err.println("Error: No se encuentra el fichero");  
    } catch(IOException e2) {  
        System.err.println("Error leyendo/escribiendo fichero");  
    }  
}
```



# Uso de flujos de procesamiento

```
public void escribe_fichero() {  
    FileWriter out = null;  
    PrintWriter p_out = null;  
    try {  
        out = new FileWriter("result.txt");  
        p_out = new PrintWriter(out);  
        p_out.println("Este texto será escrito en el fichero");  
    } catch(IOException e) {  
        System.err.println("Error al escribir en el fichero");  
    } finally {  
        p_out.close();  
    }  
}
```



# Sistema de ficheros

- La clase `File` contiene utilidades para trabajar con el sistema de ficheros
    - Constantes para indicar los separadores de directorios ('/' ó '\')
- Hace las aplicaciones independientes de la plataforma
- Crear, borrar o renombrar ficheros y directorios
  - Listar los ficheros de un directorio
  - Comprobar y establecer los permisos sobre ficheros
  - Obtener la ruta de un fichero
  - Obtener datos sobre ficheros (tamaño, fecha, etc)
  - Etc...





# Acceso a recursos

- Los recursos incluidos en un JAR no se encuentran directamente en el sistema de ficheros
  - No podremos utilizar los objetos anteriores para acceder a ellos
- Accedemos a un recurso en el JAR con

```
getClass().getResourceAsStream("/datos.txt");
```

- Anteponiendo '/' se busca de forma relativa al raíz del JAR
- Si no, buscará de forma relativa al directorio correspondiente al paquete de la clase actual



# Codificación

- Podemos codificar de forma sencilla los datos para enviarlos a través de un flujo de bytes (en serie)
- Utilizaremos un flujo `DataOutputStream`

```
String nombre = "Jose";  
int edad = 25;  
  
ByteArrayOutputStream baos = new ByteArrayOutputStream();  
  
DataOutputStream dos = new DataOutputStream(baos);  
dos.writeUTF(nombre);  
dos.writeInt(edad);  
  
dos.close();  
baos.close();  
  
byte [] datos = baos.toByteArray();
```



# Descodificación

- Para descodificar estos datos del flujo realizaremos el proceso inverso
- Utilizamos un flujo `DataInputStream`

```
ByteArrayInputStream bais = new ByteArrayInputStream(datos);
DataInputStream dis = new DataInputStream(bais);

String nombre = dis.readUTF();
int edad = dis.readInt();

dis.close();
bais.close();
```



# Entrada/Salida de objetos

- Si queremos enviar un objeto a través de un flujo deberemos convertirlo a una secuencia de bytes
- Esto es lo que se conoce como *serialización*
- Java serializa automáticamente los objetos
  - Obtiene una codificación del objeto en forma de array de bytes
  - En este array se almacenarán los valores actuales de todos los campos del objeto serializado



# Objetos serializables

- Para que un objeto sea serializable debe cumplir:

- Implementar la interfaz `Serializable`

```
public MiClase implements Serializable {  
    ...  
}
```

Esta interfaz no obliga a definir ningún método, sólo marca el objeto como serializable

- Todos los campos deben ser

Datos elementales o

Objetos serializables



# Flujos de objetos

- Para enviar o recibir objetos tendremos los flujos de procesamiento
  - `ObjectInputStream`
  - `ObjectOutputStream`
- Estos flujos proporcionan respectivamente los métodos
  - `readObject`
  - `writeObject`
- Con los que escribir o leer objetos del flujo
  - Utilizan la serialización de Java para codificarlos y decodificarlos



# Métodos de acceso a la red

- En Java podemos comunicarnos con máquinas remotas de diferentes formas
  - Mediante sockets
    - Bajo nivel
    - Problemas con firewalls intermedios
    - No adecuado para aplicaciones web
  - Acceso a URLs
    - Intercambia contenido utilizando protocolos estándar (p.ej. HTTP)
    - Java ofrece facilidades para trabajar con estos protocolos
    - No es necesario implementar los protocolos manualmente
    - Amigable con firewalls



# Acceso a alto nivel

- Encontramos también métodos de acceso remoto de alto nivel
  - Objetos distribuidos RMI/CORBA
    - Invocación de métodos remotos
    - Problemas con firewalls
    - RMI sólo accede a objetos Java
  - Servicios web
    - Permite invocar operaciones remotas
    - Protocolos web estándar, amigable con firewalls
    - Independiente del lenguaje y de la plataforma





# URLs

- URL = *Uniform Resource Locator*
  - Cadena para localizar los recursos en Internet
- Se compone de

`protocolo://servidor[:puerto]/recurso`

- P.ej. `http://www.ua.es/es/index.html`
  - Se conecta al servidor `www.ua.es`
  - A través del puerto por defecto (puerto 80)
  - Utilizando protocolo HTTP para comunicarse
  - Solicita el recurso `/es/index.html`



# URLs en Java

- Se encapsulan en la clase URL

```
URL url = new URL("http://www.ua.es/es/index.html");
```

- Es obligatorio especificar el protocolo
  - P.ej. `www.ua.es` es una URL mal formada
- Si la URL está mal formada se producirá una excepción `MalformedURLException`

```
try {  
    URL url = new URL("http://www.ua.es/es/index.html");  
} catch(MalformedURLException e) {  
    System.err.println("Error: URL mal construida");  
}
```



# Lectura del contenido

- Podemos leer el contenido de la URL abriendo un flujo de entrada con

```
InputStream in = url.openStream();
```

- Leeremos de este flujo de la misma forma que con cualquier otro flujo
  - Con los métodos a bajo nivel (byte a byte)
  - O utilizando un flujo de procesamiento
- P.ej, si la URL corresponde a un documento HTML obtendremos el código fuente de este documento



# Conexión con una URL

- Para poder tanto enviar como recibir datos debemos abrir una conexión con la URL

```
URLConnection con = url.openConnection();
```

- Creará un tipo de conexión adecuado para la URL a la que accedemos
  - P.ej, si la URL es `http://www.ua.es` creará una conexión de tipo `HttpURLConnection`
- Si vamos a enviar datos, activaremos la salida

```
con.setDoOutput(true);
```



# Estados de la conexión

- Configuración
  - Se encuentra en este estado al crearla
  - No se ha establecido la conexión
  - Podemos configurar los parámetros de la conexión
- Conectado
  - Se ha establecido la conexión
  - Podemos interactuar con el recurso al que accedemos
  - Se pasa a este estado cuando intentamos acceder a información sobre el recurso
- Cerrado
  - Se ha cerrado la conexión



# Configuración

- Podemos establecer una serie de propiedades
- Estas propiedades se enviarán al servidor al realizar la conexión
- Son parejas *<clave, valor>*

```
con.setRequestProperty(nombre, valor);
```

- Por ejemplo

```
con.setRequestProperty("IF-Modified-Since",  
                        "22 Sep 2002 08:00:00 GMT");  
con.setRequestProperty("Content-Language", "es-ES");
```



# Leer y enviar contenido

- Podemos abrir un flujo de salida para enviar contenido al servidor (si hemos activado la salida)

```
OutputStream out = con.getOutputStream();
```

- Podemos abrir un flujo de entrada para leer el contenido devuelto

```
InputStream in = con.getInputStream();
```

- Al abrir estos flujos se pasa automáticamente a estado conectado



# Cabeceras de la respuesta

- Además del contenido podemos obtener cabeceras con información sobre el recurso

```
String valor = con.getHeaderField(nombre);
```

- Cabeceras estándar:

getLength	Tamaño del contenido
getType	Tipo MIME del contenido
getEncoding	Codificación del contenido
getExpiration	Fecha de caducidad
getDate	Fecha del envío
getLastModified	Fecha de última modificación

- Leer estas cabeceras provoca el paso a estado conectado





# Ejemplo

```
// Creamos la URL y la conexión activando la salida
URL url = new URL("http://j2ee.ua.es/chat/enviar");
URLConnection con = url.openConnection();
con.setDoOutput(true);

// Escribimos los datos en un buffer en memoria
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream dos = new DataOutputStream(baos);
dos.writeUTF(nick);
dos.writeUTF(msg);
dos.close();

// Establecemos las propiedades de tipo y tamaño del contenido
con.setRequestProperty("Content-Length", String.valueOf(baos.size()));
con.setRequestProperty("Content-Type", "application/octet-stream");

// Abrimos el flujo de salida para enviar los datos al servidor
OutputStream out = con.getOutputStream();
baos.writeTo(out);

// Abrimos el flujo de entrada para leer la respuesta obtenida
InputStream in = con.getInputStream();
```

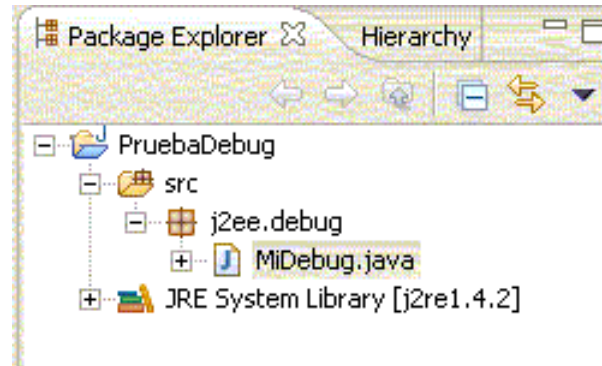


# El depurador de Eclipse

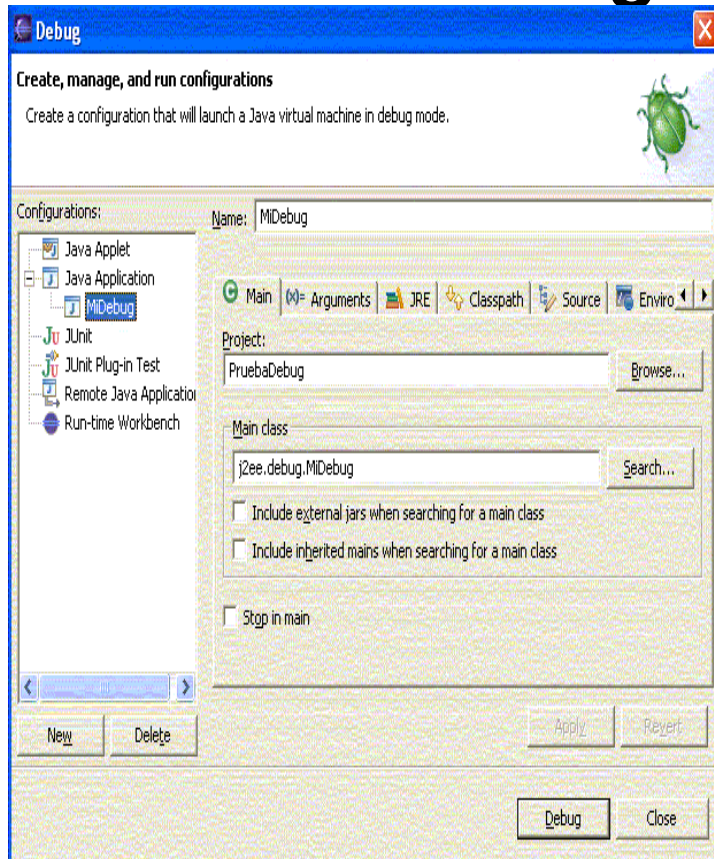
- Eclipse incorpora un depurador que permite inspeccionar cómo funciona nuestro código
- Incorpora varias funcionalidades:
  - Establecimiento de *breakpoints*
  - Consulta de valores de variables en cualquier momento
  - Consulta de valores de expresiones complejas
  - Parada/Reanudación de hilos de ejecución
- Existe también la posibilidad de depurar otros lenguajes (C/C++), instalando los plugins adecuados
- Desde Java 1.4 permite cambiar código “en caliente” y seguir con la depuración

# Paso 1: un proyecto compilado

- Para poder probar el depurador, deberemos tener ya nuestro proyecto hecho y correctamente compilado

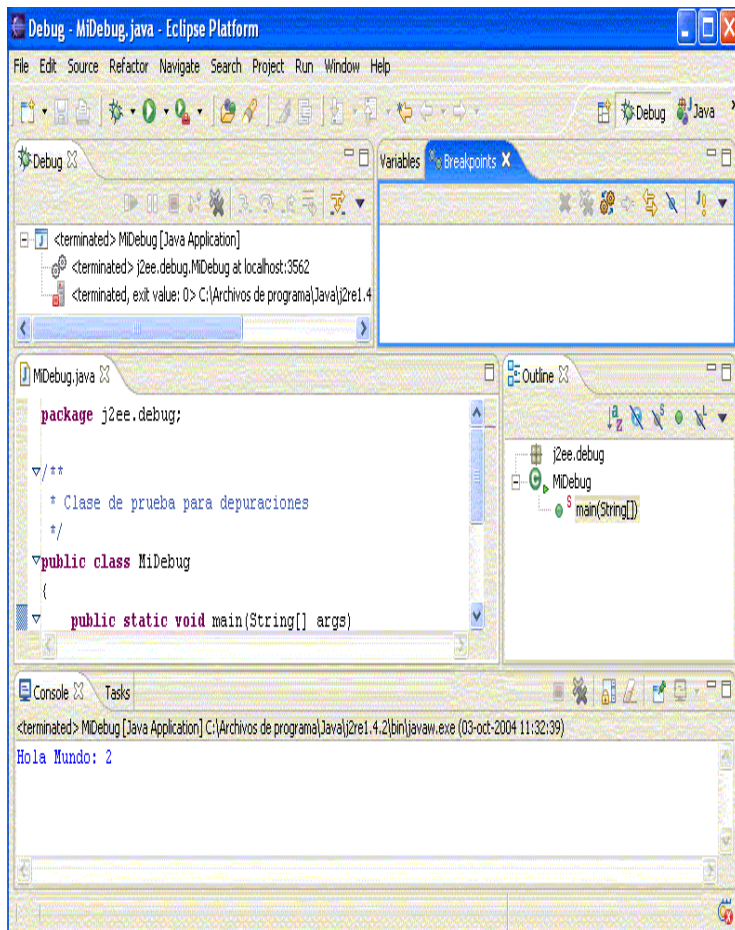


## Paso 2: configurar la depuración



- Vamos a *Run – Debug* y creamos (*New*) una nueva configuración de depuración, estableciendo la clase principal a probar
- Podremos tener tantas configuraciones sobre un proyecto como necesitemos (tantas como clases principales, normalmente)
- Pulsando en *Debug* pasaremos a depurar el código. Pulsando en *Close* cerramos la configuración

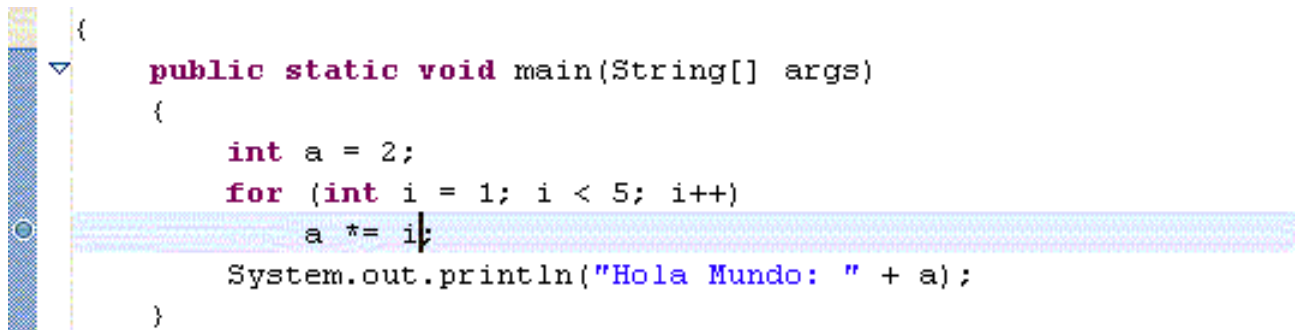
## Paso 3: ir a perspectiva de depuración



- Normalmente al depurar pasamos a la perspectiva de depuración
- Si no es así, vamos a *Window – Open Perspective – Debug*
- Vemos los hilos que se ejecutan, los breakpoints establecidos, las variables que entran en juego... etc

# Establecer *breakpoints*

- Un *breakpoint* es un punto donde la ejecución del programa se detiene para examinar su estado
- Para establecerlos, hacemos doble click en el margen izquierdo de la línea donde queremos ponerlo
- Luego arrancamos el programa desde *Run - Debug*

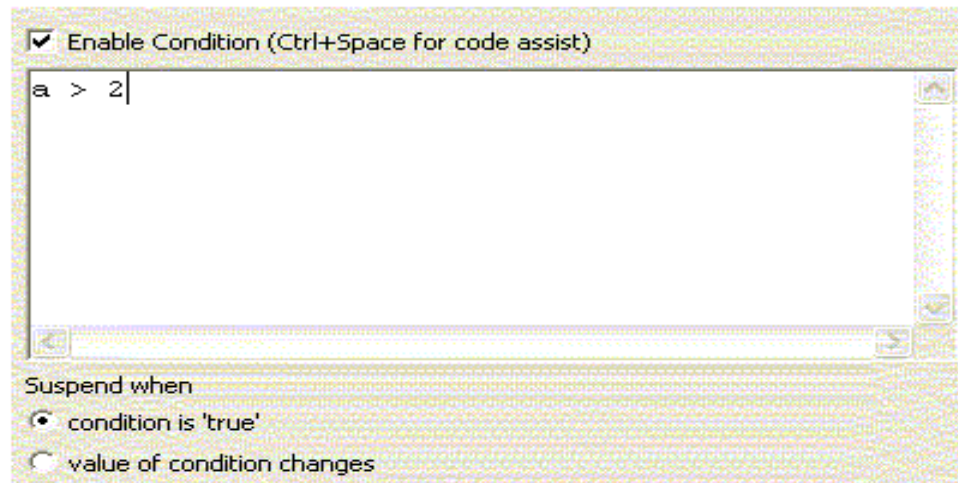


```
{  
    public static void main(String[] args)  
    {  
        int a = 2;  
        for (int i = 1; i < 5; i++)  
            a *= i;  
        System.out.println("Hola Mundo: " + a);  
    }  
}
```



# Breakpoints condicionales

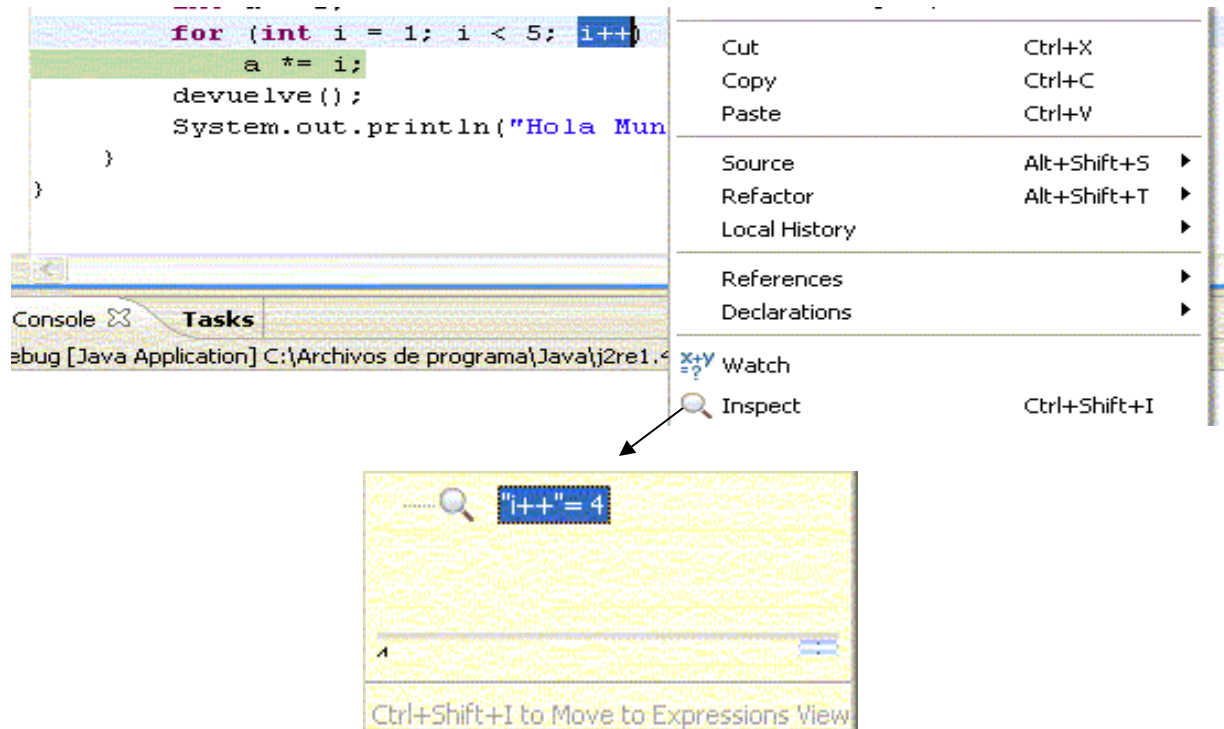
- Se disparan sólo cuando se cumple una determinada condición
- Se establecen con el botón derecho sobre el *breakpoint*, eligiendo *Breakpoint Properties*
- Colocamos la condición en *Enable Condition*





# Evaluar expresiones

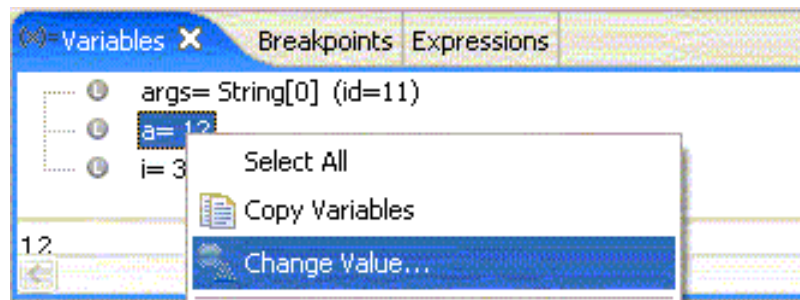
- Podemos ver el valor de una expresión compleja seleccionándola (durante una parada por *breakpoint*) y eligiendo con el botón derecho *Inspect*





# Explorar variables

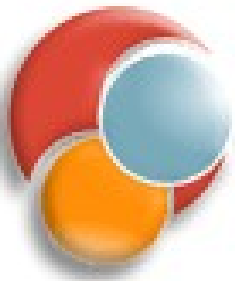
- Si queremos ver qué valores va tomando una variable paso a paso, una vez alcanzado un *breakpoint* vamos a *Run* y vamos dándole a *Step Over* o *F6*
- También podemos, en el cuadro de variables, pinchar sobre una y cambiar su valor





# Introducción a Log4J

- Log4Java (Log4J) es una librería *open source* que permite controlar la salida de los mensajes que generen nuestros programas
- Tiene diferentes niveles de mensajes, que se permiten monitorizar con cierta granularidad
- Es configurable en tiempo de ejecución
- Más información en:
  - <http://www.jakarta.apache.org/log4j>



# ¿Preguntas...?