Java

Este repaso a Java no es

- ni completo
- ni exhaustivo
- ni suficiente

para programar en Java.
Debe servir solamente para refrescar conocimiento ya adquerido y para animar de profundizar el estudio del lenguaje con otras fuentes, por ejemplo, con la bibliografía añadida y los manuales correspondientes.

Java

- Se destacan ciertas diferencias con C++ (otro lenguaje de programación orientado a objetos importante).
- Se comentan ciertos detalles del lenguaje que muchas veces no se perciben a primera vista.
- Se introducen los conceptos ya instrínsicos de Java para la programación concurrente.

hola mundo

El famoso *hola mundo* se programa en Java así:

```
class Hello {
  public static void main(String[] args) {
    System.out.println("Hello world");
  }
}
```

El programa principal se llama `main()` y tiene que ser declarado público y estático. No devuelve ningún valor (por eso se declara como `void`). Los parámetros de la línea de comando se pasan como un vector de cadenas de letras (`String`).

¿Qué se comenta?

Existen varias posibilidades de escribir comentarios:

`//`	comentario de línea
`/// ...`	comentario de documentación
`/* ... */`	comentario de bloque
`/** ... */`	comentario de documentación

- Se usa doxygen o javadoc para generar automáticamente la documentación. Ambos tienen unos comandos para aumentar la documentación.
- Se documenta sobre todo lo que no es obvio y las interfaces
- es decir: respuestas a preguntas del *¿Cómo?* y del *¿Por qué?*.

objetos

- Java usa (con la excepción de variables de tipos simples) exclusivamente objetos.
- Un tal objeto se define como una clase (`class`), y se puede crear varias instancias de objetos de tal clase.
- Es decir, la clase define el tipo del objeto, y la instancia es una variable que representa un objeto.

clases

Una clase contiene como mucho tres tipos de miembros:

- instancias de objetos (o de tipos simples)
- métodos (funciones)
- otras clases

No existen variables globales (como en C++) y el programa principal no es nada más que un método de una clase.

inicialización

- Los objetos en Java siempre tienen valores conocidos, es decir, los objetos (y también las variables de tipos simples) siempre están inicializados (menos variables locales).
- Si el programa no da una inicialización explícita, Java asigna el valor cero, es decir, `0`, `0.0`, `\u0000`, `false` o `null` dependiendo del tipo de la variable.
- Variables locales hay que inicializar antes de usarlas, el código se ejecuta cuando la ejecución llega a este punto.

Java, C++, C#

- Java y C++ (o C#) son hasta cierto punto bastante parecidos. (por ejemplo, en su síntaxis y gran parte de sus metodologías), aunque también existen grandes diferencias (por ejemplo, en su no–uso o uso de punteros y la gestión de memoria).
- Se resaltará algunos de las diferencias principales entre Java y C++.

tipos

- Java exige una disciplina estricta con sus tipos,
- es decir, el compilador controla siempre cuando pueda si las operaciones usadas están permitidas con los tipos involucrados.
- Si la comprobación no se puede realizar durante el tiempo de compilación, se pospone hasta el tiempo de ejecución,
- es decir, se pueden provocar excepciones que pueden provocar fallos durante la ejecución.

modificadores de clases I

Se pueden declarar clases con uno o varios de los siguientes modificadores para especificar ciertas propiedades (no existen en C++):

- `public` la clase es visible desde fuera del fichero
- `abstract` la clase todavía no está completa, es decir, no se puede instanciar objetos antes de que se hayan implementado en una clase derivada los métodos que faltan
- `final` no se puede extender la clase
- `strictfp` obliga a la máquina virtual a cumplir el estándar de IEEE para los números flotantes

modificadores de clases II

- Casi todos los entornos de desarrollo para Java permiten solamente una clase pública dentro del mismo fichero.
- Obviamente una clase no puede ser al mismo tiempo final y abstracta.
- Tampoco está permitida una clase abstracta con `strictfp`.

tipos simples I

`boolean`	o bien `true` o bien `false`
`char`	16 bit Unicode letra
`byte`	8 bit número entero con signo
`short`	16 bit número entero con signo
`int`	32 bit número entero con signo
`long`	64 bit número entero con signo
`float`	32 bit número flotante
`double`	64 bit número flotante

tipos simples II

- Solo `float` y `double` son igual como en C++.
- No existen enteros sin signos en Java (pero si en C++).
- Los tipos simples no son clases, pero existen para todos los tipos simples clases que implementan el comportamiento de ellos.
- Desde Java 5 la conversión de tipos simples a sus objetos correspondientes (y vice versa) es automático.
- Sólo hace falta escribirles con mayúscula (con la excepción de `Integer`).
- Las clases para los tipos simples proporcionan también varias constantes para trabajar con los números (por ejemplo, `NEGATIVE_INFINITY` etc.).

enumeraciones I

- hasta Java 1.4 se realizó enumeraciones así:

```
public final int MONDAY=0;
public final int TUESDAY=1;
public final int ...;
```

- a partir de Java 5 también así:

```
enum Weekdays { MONDAY, TUESDAY, ... }
```

- `enum` es una clase y automáticamente `public`, `static` y `final` (vemos en seguida)
- tienen `toString()` y `valueOf()`

enumeraciones II

- `enum` es una clase, es decir, se pueden añadir miembros y métodos
-
```
enum Coin {
  UN(1), DOS(2), CINCO(5), ...
  private final int value;
  Coin(int value) { this.value=value; }
  public int value() { return value; }
}
```

enumeraciones III

- `values()` devuelve un vector de los tipos del enumerado
- los `enum` se pueden usar en `switch`

```
Coin coin=...;
switch(coin) {
  case UN:
  case DOS:
  ...
}
```

modificadores de acceso

- `private`: accesible solamente desde la propia clase
- `package`: (o ningún modificador) accesible solamente desde la propia clase o dentro del mismo paquete
- `protected`: accesible solamente desde la propia clase, dentro del mismo paquete, o desde clases derivadas
- `public`: accesible siempre cuando la clase es visible

(En C++, por defecto, los miembros son privados, mientras en Java los miembros son, por defecto, del paquete.)

modificadores de miembros I

Modificadores de miembros siendo instancias de objetos:

- `final`: declara constantes si está delante de tipos simples (diferencia a C++ donde se declara constantes con `const`), aunque las constantes no se pueden modificar en el transcurso del programa, pueden ser calculadas durante sus construcciónes; las variables finales, aún declaradas sin inicialización, tienen que obtener sus valores como muy tarde en la fase de construcción de un objeto de la clase

modificadores de miembros II

- `static`: declara miembros de la clase que pertenecen a la clase y no a instancias de objetos, es decir, todos los objetos de la clase acceden a la misma cosa
- `transient`: excluye un miembro del proceso de conversión en un flujo de bytes si el objeto se salva al disco o se transmite por una conexión (no hay en C++)

modificadores de miembros III

- `volatile`: ordena a la máquina virtual de Java que no use ningún tipo de cache para el miembro, así es más probable (aunque no garantizado) que varios hilos vean el mismo valor de una variable; declarando variables del tipo `long` o `double` como `volatile` aseguramos que las operaciones básicas sean atómicas (este tema veremos más adelante más en detalle)

modificadores de métodos I

Modificadores de miembros siendo métodos:

- `abstract`: el método todavía no está completo, es decir, no se puede instanciar objetos antes de que se haya implementado el método en una clase derivada (parecido a los métodos puros de C++)
- `static`: el método pertenece a la clase y no a un objeto de la clase, un método estático puede acceder solamente miembros estáticos
- `final`: no se puede sobreescribir el método en una clase derivada (no hay en C++)

modificadores de métodos II

- `synchronized`: el método pertenece a una región crítica del objeto (no hay en C++)
- `native`: propone una interfaz para llamar a métodos escritos en otros lenguajes, su uso depende de la implementación de la máquina virtual de Java (no hay en C++, ahí se realiza durante el linkage)
- `strictfp`: obliga a la máquina virtual a cumplir el estándar de IEEE para los números flotantes (no hay en C++, ahí depende de las opciones del compilador)

modificadores de métodos III

- Un método abstracto no puede ser al mismo tiempo ni final, ni estático, ni sincronizado, ni nativo, ni estricto.
- Un método nativo no puede ser al mismo tiempo ni abstracto ni estricto.
- Nota que el uso de `final` y `private` puede mejorar las posibilidades de optimización del compilador, es decir, su uso deriva en programas más eficientes.

estructuras de control

Las estructuras de control son casi iguales a las de C++ (nota la extensión del `for` desde Java 5):

- `if(cond) then block`
- `if(cond) then block else block`
- `while(cond) block`
- `do block while (cond);`
- `for(expr; expr; expr) block`
- `for(type var: array) block`
- `for(type var: collection) block`
- `switch(expr) { case const: ... default: }`

Igual que en C++ se puede declarar una variable en la expresión condicional o dentro de la expresión de inicio del bucle `for`.

marcas I

Adicionalmente Java proporciona `break` con una marca que se puede usar para salir en un salto de varios bucles anidados.

```
mark:
  while(...) {
    for(...) {
      break mark;
    }
  }
```

marcas II

- También existe un `continue` con marca que permite saltar al principio de un bucle más alla del actual.
- No existe el `goto` (pero es una palabra reservada), su uso habitual en C++ se puede emular (mejor) con los `break`s y `continue`s y con las secuencias `try-catch-finally`.

operadores I

Java usa los mismos operadores que C++ con las siguientes excepciones:

- existe adicionalmente `>>>` como desplazamiento a la derecha llenando con ceros a la izquierda
- existe el `instanceof` para comparar tipos (C++ tiene un concepto parecido con `typeid`)
- los operadores de C++ relacionados a punteros no existen
- no existe el `delete` de C++
- no existe el `sizeof` de C++

operadores II

- La prioridad y la asociatividad son las mismas que en C++.
- Hay pequeñas diferencias entre Java y C++ si ciertos símbolos están tratados como operadores o no (por ejemplo, los `[]`).
- Además Java no proporciona la posibilidad de sobrecargar operadores.

palabras reservadas I

Las siguientes palabras están reservadas en Java:

```
abstract  default  if          private      this
boolean   do       implements  protected    throw
break     double   import      public       throws
byte      else     instanceof  return       transient
case      extends  int         short        try
catch     final    interface   static       void
char      finally  long        strictfp     volatile
class     float    native      super        while
const     for      new         switch
continue  goto     package     synchronized
```

palabras reservadas II

- Además las palabras `null`, `false` y `true` que sirven como constantes no se pueden usar como nombres.
- Aunque `goto` y `const` aparecen en la lista arriba, no se usan en el lenguaje.

objetos y referencias a objetos

- No se pueden declarar instancias de clases usando el nombre de la clase y un nombre para el objeto (como se hace en C++).
- La declaración

```
ClassName ObjectName
```

  crea solamente una referencia a un objeto de dicho tipo.
- Para crear un objeto dinámico en el montón se usa el operador `new` con el constructor del objeto deseado. El operador devuelve una referencia al objeto creado.

```
ClassName ObjectReference = new ClassName(...)
```

construcción por defecto

- Sólo si una clase no contiene ningún constructor Java propone un constructor por defecto que tiene el mismo modificador de acceso que la clase.
- Constructores pueden lanzar excepciones como cualquier otro método.

constructores

- Para facilitar la construcción de objetos aún más, es posible usar bloques de código sin que pertenezcan a constructores.
- Esos bloques están prepuestos (en su orden de aparencia) delante de los códigos de todos los constructores.
- El mismo mecanismo se puede usar para inicializar miembros estáticos poniendo un `static` delante del bloque de código.
- Inicializaciones estáticas no pueden lanzar excepciones.

inicialización estática

```
class ... {
  ...
  static int[] powertwo=new int[10];
  static {
    powertwo[0]=1;
    for(int i=1; i<powertwo.length; i++)
      powertwo[i]=powertwo[i-1]<<1;
  }
  ...
}
```

inicialización cruzada

- Si una clase, por ejemplo, `X`, construye un miembro estático de otra clase, por ejemplo, `Y`, y al revés, el bloque de inicialización de `X` está ejecutado solamente hasta la aparencia de `Y` cuyos bloques de inicialización recurren al `X` construido a medias.
- Nota que todas las variables en Java siempre están en cero si todavía no están inicializadas explícitamente.

recolector de memoria

- No existe un operador para eliminar objetos del montón, eso es tarea del recolector de memoria incorporado en Java (diferencia con C++ donde se tiene que liberar memoria con `delete` explícitamente).
- Para dar pistas de ayuda al recolector se puede asignar `null` a una referencia indicando al recolector que no se va a referenciar dicho objeto nunca jamás.
- Las referencias que todavía no acceden a ningún objeto tienen el valor `null`.
- Antes de ser destruido se exejuta el método `finalize()` del objeto (por defecto no hace nada).

reinterpretación de tipos

- Está permitida la conversión explícita de un tipo a otro mediante la reinterpretación del tipo (“cast”) con todas sus posibles consecuencias.
- El “cast” es importante especialmente en su variante del “downcast”, es decir, cuando se sabe que algún objeto es de cierto tipo derivado pero se tiene solamente una referencia a una de sus superclases.
- Se puede comprobar el tipo actual de una referencia con el operador `instanceof`.

```
if( refX instanceof refY ) { ... }
```

paso de parámetros I

- Se pueden pasar objetos como parámetros a métodos.
- La lista de parámetros junto con el nombre del método compone la signatura del método.
- Pueden existir varias funciones con el mismo nombre, siempre y cuando se distingan en sus signaturas. La técnica se llama sobrecarga de métodos.

paso de parámetros II

- Hasta Java 1.4 la lista de parámetros siempre era fija, no existía el concepto de listas de parámetros variables de C/C++.
- desde Java 5 si existe tal posibilidad.
- Java pasa parámetros exclusivamente por valor.
  Eso significa en caso de objetos que siempre se pasa una referencia al objeto con la consecuencia de que el método llamado puede modificar el objeto.

paso de parámetros III

- Desde Java 5 existen listas de parámetros variables
  `void func(int fixed, String... names) {...}`
- Los tres puntos `...` significan 0 o más parámetros.
- Solo el último parámetro puede ser variable.
- Se accede con el nuevo iterador `for`:
  `for(String name : names) {...}`

parámetros no modificables no existen

- No se puede evitar posibles modificaciones de un parámetro (que sí se puede evitar en C++ declarando el parámetro como `const`-referencia).
- Declarando el parámetro como `final` solamente protege la propia referencia (paso por valor).
- Entonces, no se pueden cambiar los valores de variables de tipos simples llamando a métodos y pasarles como parámetros variables de tipos simples (como es posible en C++ con referencias).
- La declaración se puede usar como indicación al usuario que se pretende no cambiar el objeto (aunque el compilador no lo garantiza).

valores de retorno

Un método termina su ejecución en tres ocaciones:

- se ha llegado al final de su código
- se ha encontrado una sentencia `return`
- se ha producido una excepción no tratada en el mismo método

Un `return` con parámetro (cuyo tipo tiene que coincidir con el tipo del método) devuelve una referencia a una variable de dicho tipo (o el valor en caso de tipos simples).

vectores

- Los vectores se declaran solamente con su límite súperior dado que el límite ínferior siempre es cero (`0`).
- El código

```
int[] vector = new int[15]
```

crea un vector de números enteros de longitud 15.

control de acceso

- Java comprueba si los accesos a vectores con índices quedan dentro de los límites permitidos (diferencia con C++ donde no hay una comprobación).
- Si se detecta un acceso fuera de los límites se produce una excepción `IndexOutOfBoundsException`.
- Dependiendo de las capacidades del compilador eso puede resultar en una pérdida de rendimiento.

vectores son objetos

- Los vectores son objetos implícitos que siempre conocen sus propias longitudes (`values.length`) (diferencia con C++ donde un vector no es nada más que un puntero) y que se comportan como clases finales.
- No se pueden declarar los elementos de un vector como constantes (como es posible en C++), es decir, el contenido de los componentes siempre se puede modificar en un programa en Java.

`this` and `super`

- Cada objeto tiene por defecto una referencia llamada `this` que proporciona acceso al propio objeto (diferencia a C++ donde `this` es un puntero).
- Obviamente, la referencia `this` no existe en métodos estáticos.
- Cada objeto (menos la clase `object`) tiene una referencia a su clase súperior llamada `super` (diferencia a C++ donde no existe, se tiene acceso a las clases superiores por otros medios).
- `this` y `super` se pueden usar especialmente para acceder a variables y métodos que están escondidos por nombres locales.

más sobre constructores

- Para facilitar las definiciones de constructores, un constructor puede llamar en su primer sentencia
  - o bien a otro constructor con `this(...)`
  - o bien a un constructor de su superclase con `super(...)` (ambos no exiten en C++).
- El constructor de la superclase sin parámetros está llamado en todos los casos al final de la posible cadena de llamadas a constructores `this()` en caso que no haya una llamada explícita.

orden de construcción

La construcción de objetos sigue siempre el siguiente orden:

- construcción de la superclase, nota que no se llama ningún constructor por defecto que no sea el constructor sin parámetros
- ejecución de todos los bloques de inicialización
- ejecución del código del constructor

extender clases I

- Se puede crear nuevas clases a partir de la extención de clases ya existentes (en caso que no sean finales). Las nuevas clases se suelen llamar subclases o clases extendidas.
- Una subclase heredará todas las propriedades de la clase súperior, aunque se tiene solamente acceso directo a las partes de la superclase declaradas por lo menos `protected`.

extender clases II

- No se puede extender al mismo tiempo de más de una clase súperior (diferencia a C++ donde se puede derivar de más de una clase).
- Se pueden sobreescribir métodos de la superclase.
- Si se ha sobreescrito una cierta función, las demás funciones con el mismo nombre (pero diferente signatura) siguen visibles desde la clase derivada (en C++ eso no es el caso).
- Dicho último aspecto puede provocar sorpresas... ¿Cuáles?

acceso a métodos sobreescritos

- Si se quiere ejecutar dentro de un método sobreescrito el código de la superclase, se puede acceder el método original con la referencia `super`.
- Se puede como mucho extender la accesibilidad de métodos sobreescritos.
- Se pueden cambiar los modificadores del método.
- También se puede cambiar si los parámetros del método son finales o no, es decir, `final` no forma parte de la signatura (diferencia a C++ donde `const` forma parte de la signatura).

sobreescritura y excepciones

- Los tipos de las excepciones que lanza un método sobreescrito tienen que ser un subconjunto de los tipos de las excepciones que lanza el método de la superclase.
- Dicho subconjunto puede ser el conjunto vacio.
- Si se llama a un método dentro de una jerarquia de clases, se ejecuta siempre la versión del método que corresponde al objeto creado (y no necesariamente al tipo de referencia dado) respetando su accesibilidad.
- Está técnica se llama polimorfismo.

clases dentro de clases

- Se pueden declarar clases dentro de otras clases.
- Sin embargo, dichas clases no pueden tener miembros estáticos no–finales.
- Todos los miembros de la clase contenedora están visibles desde la clase interior (diferencia a C++ donde hay que declarar la clase interior como `friend` para obtener dicho efecto).

clases locales

Dentro de cada bloque de código se pueden declarar clases locales que son visibles solamente dentro de dicho bloque.

la clase `Object` I

Todos los objetos de Java son extensiones de la clase `Object`. Los métodos públicos y protegidos de esta clase son

- `public boolean equals(Object obj)` compara si dos objetos son iguales, por defecto un objeto es igual solamente a si mismo
- `public int hashCode()` devuelve (con alta probabilidad) un valor distinto para cada objeto
- `protected Object clone() throws CloneNotSuportedException` devuelve una copia binaria del objeto (incluyendo sus referencias)

la clase `Object` II

- `public final Class getClass()` devuelve el objeto del tipo `Class` que representa dicha clase durante la ejecución
- `protected void finalize() throws Throwable` se usa para finalizar el objeto, es decir, se avisa al administrador de la memoria que ya no se usa dicho objeto, y se puede ejecutar código especial antes de que se libere la memoria
- `public String toString()` devuelvo una cadena describiendo el objeto

Las clases derivadas deben sobreecribir los métodos adecuadamente, por ejemplo el método `equals`, si se requiere una comparación binaria.

interfaces

- Usando `interface` en vez de `class` se define una interfaz a una clase sin especificar el código de los métodos.
- Una interfaz no es nada más que una especificación de cómo algo debe ser implementado para que se pueda usar en otro código.
- Una interfaz solo puede tener declaraciones de objetos que son constantes (`final`) y estáticos (`static`).
- En otras palabras, todas las declaraciones de objetos dentro de interfaces automáticamente son finales y estáticos, aunque no se haya descrito explícitamente.

interfaces y herencia

- Igual que las clases, las interfaces pueden incorporar otras clases o interfaces.
- También se pueden extender interfaces.
- Nota que es posible extender una interfaz a partir de más de una interfaz:

```
interface ThisOne extends ThatOne, OtherOne { ... }
```

métodos de interfaces

- Todos los métodos de una interfaz son implícitamente públicos y abstractos, aunque no se haya descrito ni `public` ni `abstract` explícitamente (y eso es la convención).
- Los demás modificadores no están permitidos para métodos en interfaces.
- Para generar un programa todas las interfaces usadas tienen que tener sus clases que las implementen.

implementación de interfaces

- Una clase puede implementar varias interfaces al mismo tiempo (aunque una clase puede extender como mucho una clase).
- Se identifican las interfaces implementadas con `implements` después de una posible extensión (`extends`) de la clase.

implementación

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

class Something extends Anything
  implements Comparable
{ ...
  public int compareTo(Object o) {
      // cast to get a correct object
      // may throw exception ClassCastException
    Something s = (Something)o;
    ... // code to compare to somethings
  }
}
```

resumen: interfaces

Las interfaces se comportan como clases totalmente abstractas, es decir,

- no tienen miembros no–estáticos,
- nada diferente a público,
- y ningún código no–estático.

tipos como variables

- Como ya existía en C++, se introdujo la posibilidad de usar tipos como variables en la definición de clases y métodos.
- Se realiza con una sintaxis parecida:
  `List<Animal> farm=new ArrayList<Animal>();`
- Con eso se evita las muchas transformaciones explícitas de tipos que antes su usaba sobre todo para agrupar objetos en colecciones.
- Es decir, se puede diseñar estructuras de datos sin especificar antemano con que tipo se trabrajá en concreto.
- Cuando se usa el compilador garantiza que el tipo concreto proporciona las propiedades necesarias.

clases genéricas

```
class Something<T> {
  T something;
  public Something(T something) {
    this.something=something;
  }
  public void set(T something) {
    this.something=something;
  }
  public T get() {
    return something;
  }
}
```

uso de clase genérica

- Usamos la clase `Something` con cadenas.
- Construcción:

```
Something<String> w=new Something<String>("word");
```

- Leer el “contenido”:

```
String s=w.get();
```

- Escribir el “contenido”:

```
w.set(new Double(10.0));
```

producerá un fallo de compilación, hay que usar una cadena como parámetro.

métodos genéricos

```
class Anything {
  public <T> T get(T something) {
    return something;
  }
  public static <T> void write(T something) {
    out.println(something);
  }
}
```

Con métodos genéricos se pueden implementar funcionalidades que se quieren realizar con cualquier tipo de interés.

polimorfismo paramétrico restringido

- Se puede declarar el tipo que se usa para especificar un tipo genérico asumiendo cierta herencia:

```
List<T extends Animal>
```

- Así en el uso del tipo `T` ya se sabe algo sobre sus funcionalidades (y el compilador lo comprueba).

polimorfismo paramétrico anidado/encadenado

- Se puede expresar también que el tipo genérico se heredera de otro tipo genérico:

```
List<T extends Animal<E>>
```

- o que el tipo genérico ya viene dado por otro tipo genérico

```
LinkedList<LinkedList<T>>
```

limitaciones del polimorfismo paramétrico

- No se puede instanciar un objeto de un tipo genérico, sino es dentro de una clase o método del mismo, es decir,
- `T e=new T();` está prohibido
- `List<T> L= new LinkedList<T>();` está permitido.

polimorfismo paramétrico con comodín I

- Observa: `List<Object>` *no* es superclase de `List<String>`.
- Entonces, para escribir métodos (y clases) que trabajen con cualquier *tipo genérico* necesitamos una notación nueva:
- `List<?>`
- sirve para implementar por ejemplo

```
void write(List<?> L) {
  for(Object e : L) out.println(e);
}
```

polimorfismo paramétrico con comodín II

- Los comodines adquieren forma en su construcción:

```
Collection<?> C = new ArrayList<String>();
```

- ahora la colección `C` contiene cadenas.
- Solo `null` se puede asignar a una variable del tipo comodín, siempre.
- Eso no funciona para pasar parámetros:

  ```
  <T> void add(Set<T> s, T t) {...}
  ```

  no se puede usar con un conjunto construido genéricamente

  ```
  add(new Set<String>(), new String("hi"));
  ```

polimorfismo paramétrico con comodín III

- Los comodines se pueden usar también para expresar la propia cadena de herencia que se quiere mantener:

```
Collection<? extends Shape> C
  = new ArrayList<Circle>();
```

- donde `Circle` tiene que ser un tipo cuya superclase es `Shape`.
- Dicho concepto se llama comodín limitado.
- Pero ya no existe la posibilidad de escribir (la relación no es reflexiva)

```
Collection<? extends Shape> C
  = new ArrayList<Shape>();
```

polimorfismo paramétrico con comodín IV

- También se puede limitar el comodín desde abajo:

```
Collection<? super Circle> C
  = new ArrayList<Shape>();
```

- Aquí sí se puede escribir

```
Collection<? super Circle> C
  = new ArrayList<Circle>();
```

dado que la relación es reflexiva.

polimorfismo paramétrico con comodín V

- Los tipos genéricos (tanto comodín o no-comodín) se transforman en tipos simples *antes* de la ejecución.
- Por eso no se tiene acceso a la variable del tipo, con la consecuencia que

```
List<String> S=new ArrayList<String>();
List<Integer> I=new ArrayList<Integer>();
out.println(S.getClass()==I.getClass());
```

  imprime `true`.
- Tampoco se puede averiguar el tipo, el siguiente código no compila:

```
Collection<String> S=new ArrayList<String>();
if(S instanceof Collection<String>) \{...\}
```

polimorfismo paramétrico con comodín VI

- Hay que tomarse muy en serio posibles mensajes de aviso cuando se usa tipos genéricos y cambiar el código hasta que no aparezca ninguno.
- Sino, puede ocurrir una simple excepción de fallo en conversión de tipo en algún momento de la ejecución cuya razón será difícil de localizar.

try'n'catch

- Para facilitar la programación de casos excepcionales Java usa el concepto de lanzar excepciones.
- Una excepción es una clase predefinida y se accede con la sentencia

```
try { ... }
catch (SomeExceptionObject e) { ... }
catch (AnotherExceptionObject e) { ... }
finally { ... }
```

orden de ejecución I

- El bloque `try` contiene el código normal por ejecutar.
- Un bloque `catch(ExceptionObject)` contiene el código excepcional por ejecutar en caso de que durante la ejecución del código normal (que contiene el bloque `try`) se produzca la excepción del tipo adecuado.
- Pueden existir más de un (o ningún) bloque `catch` para reaccionar directamente a más de un (ningún) tipo de excepción.
- Hay que tener cuidado en ordenar las excepciones correctamente, es decir, las más específicas antes de las más generales.

orden de ejecución II

- El bloque `finally` se ejecuta siempre una vez terminado o bien el bloque `try` o bien un bloque `catch` o bien una excepción no tratada o bien antes de seguir un `break`, un `continue` o un `return` hacia fuera de la sentencia `try-catch-finally`.

construcción de clases de excepción

Normalmente se extiende la clase `Exception` para implementar clases propias de excepciones, aún también se puede derivar directamente de la clase `Throwable` que es la superclase (interfaz) de `Exception` o de la clase `RuntimeException`.

```
class MyException extends Exception {
  public MyException() { super(); }
  public MyException(String s) { super(s); }
}
```

declaración de excepciones lanzables

- Entonces, una excepción no es nada más que un objeto que se crea en el caso de aparición del caso excepcional.
- La clase principal de una excepción es la interfaz `Throwable` que incluye un `String` para mostrar una línea de error legíble.
- Para que un método pueda lanzar excepciones con las sentencias `try-catch-finally`, es imprecindible declarar las excepciones posibles antes del bloque de código del método con `throws ...`.

```
public void myfunc(...) throws MyException {...}
```

- En C++ es al revés, se declara lo que se puede lanzar como mucho.

propagación de excepciones

- Durante la ejecución de un programa se propagan las excepciones desde su punto de aparición subiendo las invocaciones de los métodos hasta que se haya encontrado un bloque `catch` que se ocupa de tratar la excepción.
- En el caso de que no haya ningún bloque responsable, la excepción será tratada por la máquina virtual con el posible resultado de abortar el programa.

lanzar excepciones

- Se pueden lanzar excepciones directamente con la palabra `throw` y la creación de un nuevo objeto de excepción, por ejemplo:

```
throw new MyException("eso es una excepcion");
```

- También los constructores pueden lanzar excepciones que tienen que ser tratados en los métodos que usan dichos objetos construidos.

excepciones de ejecución

- Además de las excepciones así declaradas existen siempre excepciones que pueden ocurrir en qualquier momento de la ejecución del programa, por ejemplo, `RuntimeException` o `Error` o `IndexOutOfBoundException`.
- La ocurrencia de dichas excepciones refleja normalmente un flujo de control erróneo del programa que se debe corrigir antes de distribuir el programa a posibles usuarios.
- Se usan excepciones solamente para casos excepcionales, es decir, si pasa algo no esperado.

agrupación de objetos I

- Siempre existe la posibilidad de que diferentes fuentes usen el mismo nombre para una clase.
- Para producir nombres únicos se agrupa los objetos en paquetes.
- El nombre del paquete sirve como prefijo del nombre de la clase con la consecuencia de que cuando se diferencian los nombres de los paquetes también se diferencian los nombres de las clases.

agrupación de objetos II

- Por convención se usa como prefijo el dominio en internet en orden inverso para los paquetes.
- Hay que tener cuidado en distinguir los puntos en el nombre del paquete con los puntos que separan los miembros de una clase.
- La pertenencia de una clase a un paquete se indica en la primera sentencia de un fichero fuente con

```
package Pack.Name;
```

agrupación de objetos III

- Java viene con una amplia gama de clases y paquetes predefinidos.
- Se accede a los paquetes con `import`.
- Se accede a los componentes de los paquetes con clasificadores, p.ej., `System.out.println(...)` (desde Java 5 ya no hace falta clasificar, se importa como `import static java.lang.System.*`).
- Cuidado: Java no está disponible siempre en todas las plataformas en su última versión y eso puede derivar en aplicaciones no portables.

acceso a si mismo

- Java proporciona para cada clase un objeto de tipo `Class` que se puede usar para obtener información sobre la propia clase y todos sus miembros.
- Así por ejemplo se puede averiguar todos los métodos y modificadores, cual es su clase súperior y mucho más.

objetivos

Se usan los hilos para ejecutar varias secuencias de instrucciones de modo cuasi–paralelo.

creación de un hilo (para empezar)

- Se crea un hilo con

```
Thread worker = new Thread()
```

- Después se inicializa el hilo y se define su comportamiento.
  Se lanza el hilo con

```
worker.start()
```

- Pero en esta versión simple no hace nada. Hace falta sobreescribir el método `run()` especificando algún código útil.

la interfaz `Runnable`

- A veces no es conveniente extender la clase `Thread` porque se pierde la posibilidad de extender otro objeto.
- Es una de las razones por que existe la interfaz `Runnable` que declara nada más que el método `public void run()` y que se puede usar fácilmente para crear hilos trabajadores.

pingPONG I

```
class RunPingPONG implements Runnable {
  private String word;
  private int delay;

  RunPingPONG(String whatToSay, int delayTime) {
    word =whatToSay;
    delay=delayTime;
  }
```

pingPONG II

```
public void run() {
  try {
    for(;;) {
      System.out.print(word+" ");
      Thread.sleep(delay);
    }
  }
  catch(InterruptedException e) {
    return;
  }
}
```

pingPONG III

```
  public static void main(String[] args) {
    Runnable ping = new RunPingPONG("ping", 40);
    Runnable PONG = new RunPingPONG("PONG", 50);
    new Thread(ping).start();
    new Thread(PONG).start();
  }
}
```

construcción de `Runnable`s

Existen cuatro constructores para crear hilos usando la interfaz `Runnable`.

- `public Thread(Runnable target)`
  así lo usamos en el ejemplo arriba, se pasa solamente la implementación de la interfaz `Runnable`
- `public Thread(Runnable target, String name)`
  se pasa adicionalmente un nombre para el hilo
- `public Thread(ThreadGroup group, Runnable target)`
  construye un hilo dentro de un grupo de hilos
- `public Thread(ThreadGroup group, Runnable target, String name)`
  construye un hilo con nombre dentro de un grupo de hilos

implementación de `Runnable`

- La interfaz `Runnable` exige solamente el método `run()`, sin embargo, normalmente se implementan más métodos para crear un servicio completo que este hilo debe cumplir.
- Aunque no hemos guardado las referencias de los hilos en unas variables, los hilos *no caen* en las manos del recolector de memoria: siempre se mantiene una referencia al hilo en su grupo al cual pertenece.
- El método `run()` es público y en muchos casos, implementando algún tipo de servicio, no se quiere dar permiso a otros ejecutar directamente el método `run()`. Para evitar eso se puede recurrir a la siguiente construcción:

`run()` no-público

```
class Service {
  private Queue requests = new Queue();
  public Service() {
    Runnable service = new Runnable() {
      public void run() {
        for(;;) realService((Job)requests.take());
      }
    };
    new Thread(service).start();
  }
  public void AddJob(Job job) {
    requests.add(job);
  }
  private void realService(Job job) {
    // do the real work
  }
}
```

explicación del ejemplo

- Crear el servicio con `Service()` lanza un nuevo hilo que actua sobre una cola para realizar su trabajo con cada tarea que encuentra ahí.
- El trabajo por hacer se encuentra en el método privado `realService()`.
- Una nueva tarea se puede añadir a la cola con `AddJob(...)`.
- **Nota:** la construcción arriba usa el concepto de clases anónimas de Java, es decir, sabiendo que no se va a usar la clase en otro sitio que no sea que en su punto de construcción, se declara directamente donde se usa.

sincronización

- En Java es posible forzar la ejecución del código en un bloque en modo sincronizado, es decir, como mucho un hilo puede ejecutar algún código dentro de dicho bloque al mismo tiempo.

```
synchronized (obj) { ... }
```

- La expresión entre paréntesis `obj` tiene que evaluar a una referencia a un objeto o a un vector.
- Declarando un método con el modificador `synchronized` garantiza que dicho método se ejecuta ininterrumpidamente por un sólo hilo.
- La máquina virtual instala un cerrojo (mejor dicho, un monitor, ya veremos dicho concepto más adelante) que se cierra de forma atómica antes de entrar en la región crítica y que se abre antes de salir.

métodos syncronizados

- Declarar un método como

  ```
  synchronized void f() { ... }
  ```

  es equivalente a usar un bloque sincronizado en su interior:

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

- Los monitores permiten que el mismo hilo puede acceder a otros métodos o bloques sincronizados del mismo objeto sin problema.
- Se libera el cerrojo sea el modo que sea que termine el método.
- Los constructores no se pueden declarar `synchronized`.

syncronización y herencia

- No hace falta mantener el modo sincronizado sobreescribiendo métodos síncronos mientras se extiende una clase. (No se puede *forzar* un método sincronizada en una interfaz.)
- Sin embargo, una llamada al método de la clase súperior (con `super.`) sigue funcionando de modo síncrono.
- Los métodos estáticos también pueden ser declarados `synchronized` garantizando su ejecución de manera exclusiva entre varios hilos.

protección de miembros estáticos

En ciertos casos se tiene que proteger el acceso a miembros estáticos con un cerrojo. Para conseguir eso es posible sincronizar con un cerrojo de la clase, por ejemplo:

```
class MyClass {
  static private int nextID;
  ...
  MyClass() {
    synchronized(MyClass.class) {
      idNum=nextID++;
    }
  }
  ...
}
```

¡Ojo con el concepto!

Declarar un bloque o un método como síncrono solo prevee que ningún otro hilo pueda ejecutar al mismo tiempo dicha región crítica, sin embargo, cualquier otro código asíncrono puede ser ejecutado mientras tanto y su acceso a variables críticas puede dar como resultado fallos en el programa.

objetos síncronos

Se obtienen objetos totalmente sincronizados siguiendo las reglas:

- todos los métodos son `synchronized`,
- no hay miembros/atributos públicos,
- todos los métodos son `final`,
- se inicializa siempre todo bien,
- el estado del objeto se mantiene siempre consistente incluyiendo los casos de excepciones.

páginas del manual

Se recomienda estudiar detenidamente las páginas del manual de Java que estén relacionados con el concepto de hilo.

atomicidad en Java

- Solo las asignaciónes a variables de tipos simples de 32 bits son atómicas.
- `long` y `double` no son simples en este contexto porque son de 64 bits, hay que declararlas `volatile` para obtener acceso atómico.

limitaciones para la programación concurrente

- no se puede interrumpir la espera a un cerrojo (una vez llegado a un `synchronized` no hay vuelta atrás)
- no se puede influir mucho en la política del cerrojo (distinguir entre lectores y escritores, diferentes justicias, etc.)
- no se puede confinar el uso de los cerrojos (en cualquier línea se puede escribir un bloque sincronizado de cualquier objeto)
- no se puede adquirir/liberar un cerrojo en diferentes sitios, se está obligado a un estructura de bloques

paquete especial para la programación concurrente

- Por eso se ha introducido desde Java 5 un paquete especial para la programación concurrente.

  `java.util.concurrent`

- Hay que leer todo su manual.

resumen respecto a concurrencia

- (transient)
- volatile
- synchronized
- try-catch-finally
- finalize
- Thread, Runnable
- java.util.concurrent
- java.util.concurrent.atomic