U.D. 5 - Compilación, enlace y gestión de memoria en lenguajes OO

- Vamos a hablar de los tipos de datos en los lenguajes de programación porque la manera en la que cada lenguaje implementa lo que se conoce como **sistemas de tipos** tiene una gran relevancia en el diseño del lenguaje (o a la inversa) y en las decisiones que hay que tomar durante la ejecución de los programas.
- Lee el siguiente texto (en inglés), que servirá como introducción a los conceptos que vamos a desarrollar en este tema:

"In programming languages, a **type system** is a collection of rules that assign a property called **type** to various constructs a computer program consists of, such as variables, expressions, functions or modules. The main purpose of a type system is to reduce possibilities for bugs in computer programs by defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way. This checking can happen **statically** (at compile time), **dynamically** (at run time), or as a combination of **static and dynamic checking**. Type systems have other purposes as well, such as enabling certain compiler optimizations, providing a form of documentation, etc. A type system associates a type with each computed value and, by examining the flow of these values, attempts to ensure or prove that no **type errors** can occur. The particular type system in question determines exactly what constitutes a type error, but in general the aim is to prevent operations expecting a certain kind of value from being used with values for which that operation does not make sense". (Wikipedia)

- También hablaremos de cómo funciona un compilador y del concepto relacionado de intérprete.

Tipos

- Consideremos un lenguaje como Java en el que cada dato tiene un **tipo** y cada expresión también.
- En el diagrama de la figura 1 vemos que cada dato tiene un tipo. Cuando se aplican operadores la expresión resultante tiene un determinado tipo que se deduce de la aplicación de ciertas reglas que dependen del **sistema de tipos** de cada lenguaje.
- Como vemos en una expresión se pueden combinar distintos tipos y se hacen conversiones implícitas cuando corresponde. Cada lenguaje es más o menos estricto a la hora de permitir combinar diversos tipos mediante operadores binarios o aplicar ciertos operadores unarios a datos de un determinado tipo.

Lenguajes fuertemente tipados y lenguajes debilmente tipados

- Estos términos no tienen una definición precisa pero, en general, un lenguaje fuertemente tipado generará un error o sencillamente no compilará si dos tipos de datos (por ejemplo, en una asignación o al pasar un argumento a una función) no son el mismo o compatibles entre sí. Por el contraro, un lenguaje muy debilmente tipado aceptará la situación a riesgo de provocar resultados impredecibles en ejecución, o aplicando conversión implícita de tipos.
- Son términos que no tienen una definición precisa y consensuada pero nos sirven para nuestra discusión. Pero que quede claro que unos lenguajes tienen tipos más fuertes que otros sin que se pueda hablar de si un lenguaje tiene tipos fuertes o no. A veces depende del tipo de dato u

operación de manera que un lenguaje puede ser más fuerte que otro a veces y otras veces no.

- "Is C# a strongly typed or a weakly typed language? Yes" Fuente.
- En Pascal no podemos poner una expresión no booleana en un condicional ni sumar un puntero y un entero. En C, por ejemplo, sí se puede.
- En C++ no podemos concatenar una cadena y un entero ("foo"+5); en Java, sí. Ojo, de todos modos con esto:

```
int x=20,y=10;

System.out.println("printing: " + x + y); // printing: 2010

System.out.println("printing: " + x * y); // printing: 200
```

- Como cursiosidad, en C++ tendríamos que hacer algo como:

```
"printing: " + std::to_string(x+y);
```

Lenguajes estáticos y dinámicos

- Otra dimensión de estudio diferente es la de si las variables han de tener un tipo cuando se declaran.
- En Java:

```
int a;
a= 5;
a= "foo";  <- error del compilador
```

- En JavaScript:

```
var a;
a= 5;
a= "foo"; // ok
```

- Esto nos lleva a otra caracterización de los lenguajes como **estáticos** o **dinámicos**. En este ejemplo, Java se considera un lenguaje estático, mientras que JavaScript es un lenguaje dinámico. De nuevo, se trata de un continuo y ni siquiera se puede decir que un lenguaje es más dinámico que otro siempre sino que puede depender de qué aspecto concreto estamos analizando.
- Suele haber cierta correlación entre lenguajes de tipos fuertes/débiles y lenguajes estáticos/dinámicos, aunque no es una consecuencia directa.
- Consideremos este ejemplo en C++ y luego en JavaScript:

```
using namespace std;
...
int a;
int b;
string s;
...
int c= a+b;
string t= s+b;
```

- En este ejemplo, el compilador decide por el sistema de tipos que la última operación no es correcta. No nos hace falta ejecutar el programa; de hecho, no podemos porque no hay ejecutable generado. C++ se comporta como un lenguaje estático aquí: se comprueban los tipos en **tiempo de compilación**.
- En un lenguaje dinámico no queda más remedio que esperar a la ejecución del programa. Por ejemplo, en JavaSCript:

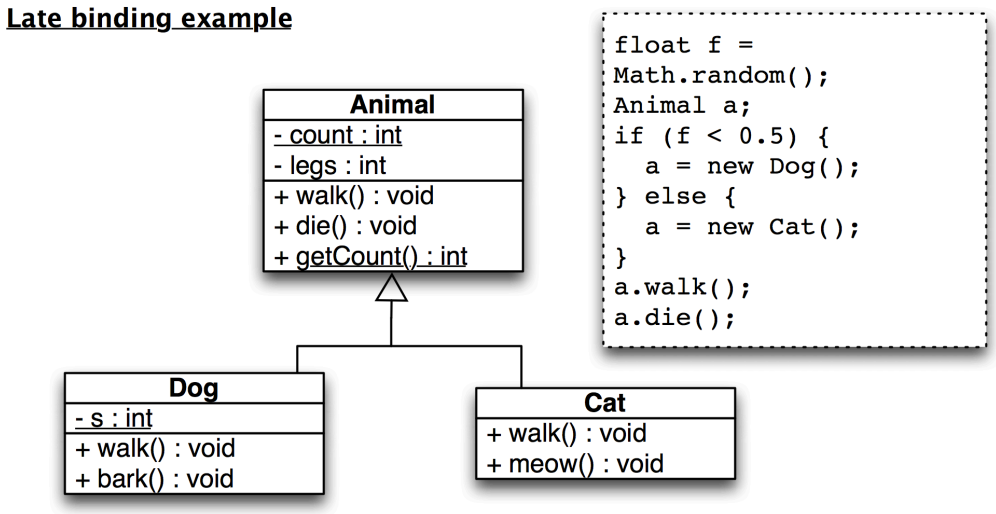
```
var a,b;
if (Math.random() < 0.5) {
  a= "foo";
  b= "bar";
} else {
  a= 5;
  b= 3;
}
var c= a+b;  // no sabríamos qué código generar porque no sabríamos los tipos en tiempo de compilación
```

- Lenguajes como JavaScript, que no realizan comprobaciones de tipo en tiempo de compilación, tienen que esperar a la ejecución del programa (**tiempo de ejecución**) para conocer muchos de los tipos. En Java, sin embargo, como en C++, se conocen en tiempo de compilación, aunque más adelante veremos que incluso Java (y C++) retardan algunas decisiones de tipos a la ejecución del programa.
- Por lo anterior los **lenguajes dinámicos** se suelen **interpretar** y no se compilan (no hay ejecutable). Diríamos que hay tanto que hacer durante la ejecución del programa que se necesita un entorno/máquina virtual/motor/intérprete para ir haciendo estas cosas.
- Un **intérprete** de código ejecuta éste directamente, sin generar un ejecutable. Básicamente, se lee cada línea del fichero de texto con el código fuente y se hace algo con ella (se ejecuta).

Compiladores e intérpretes

- Para entender mejor la diferencia entre intérpretes y compiladores es necesario saber cómo funciona un compilador.
- En el ejemplo de la figura 2 podemos ver las fases de un compilador al compilar una instrucción:
    - **Análisis léxico**: la instrucción `int a = 3+5*2` se analiza para convertir cada *token* de la entrada en un símbolo reconocido por el compilador: 'int', 'ID', 'ENTERO'.
    - **Análisis sintáctico (parsing)**: La cadena de símbolos generada por el analizador léxico se analiza sintácticamente para ver si concuerda con alguna estructura del lenguaje (en este caso, una asignación).

    - **Análisis semántico** (por ejemplo, tipos de datos correctos, como vimos antes): A medida que se analiza sintácticamente la cadena de símbolos, se van comprobando cosas como la compatibilidad de tipos entre los operadores de las distintas oepraciones que se realizan (suma, multiplicacion, asignación). Si, por ejemplo, el tipo resultante de la expresión en la parte derecha de la asignación no es compatible con el tipo del identificador al que se quiere asignar el resultado, se emite un error de compilación.
    - **Generación de código**: Conforme se van analizando partes de la instrucción que son correctas, se va generando código (en el ejemplo, en verde, código ensamblador) que se va 'montando' conforme el análisis va construyendo el árbol sintáctico completo de la instrucción. En cada nivel, usamos el código que sube de abajo y se añade quizás algo nuevo.
- Debemos tener en cuenta que entre la generación del código ejecutable y su ejecución pueden pasar segundos o incluso años.
- El intérprete es parecido pero las tres fases de análisis ocurren justo antes de la ejecución en sí o, mejor dicho, se va analizando el programa mientras se ejecuta. La fase de generación se sustituye por la ejecución de cada instrucción. En un intérprete no existe esa disociación entre el análisis, la generación y la ejecución, sino que todo ocurre a la vez.
- Pero, de nuevo, no todo es blanco o negro:
    1. A diferencia de C++, un compilador de Java genera *bytecode*, que puede considerarse como un lenguaje a medio camino entre un lenguaje de alto nivel y uno de bajo nivel, como ensamblador. Este bytecode, de hecho, es interpretado por la Máquina Virtual de Java (*Java Virtual Machine, JVM*).
    2. Los intérpretes a veces pasan el código fuente a código máquina (por ejemplo, el compilador *just in time* de los motores de JavaScript de los navegadores), para mejorar el rendimiento de manera que una función no se interpreta cada vez que se ejecuta sino sólo antes de su primera llamada.
    3. La misma JVM (aunque no es requisito del lenguaje) suele tener un compilador *just in time* para pasar el bytecode a código máquina.
    4. Los compiladores de algunos lenguajes, como ya anticipamos antes, no conocen siempre los tipos de todo lo que aparece en el código fuente. Es decir, los lenguajes no son necesariamente siempre estáticos o siempre dinámicos. En Java parte de la dinamicidad viene por lo que se conoce como *polimorfismo*.
- Veamos un ejemplo del último caso en Java. Fíjate en el siguiente diagrama UML. El código a su lado define una *variable polimórfica*.



- En este ejemplo el compilador no puede saber el tipo exacto de *a* y, por tanto, no sabe tampoco en qué dirección de la memoria de código tiene que saltar para ejecutar *walk* (supongamos que en Java se hace algo parecido a un *call* de ensamblador a una dirección de memoria, aunque sabemos que no es así). Por lo tanto, a la pregunta de si Java es un lenguaje estático o dinámico la respuesta es *sí*.

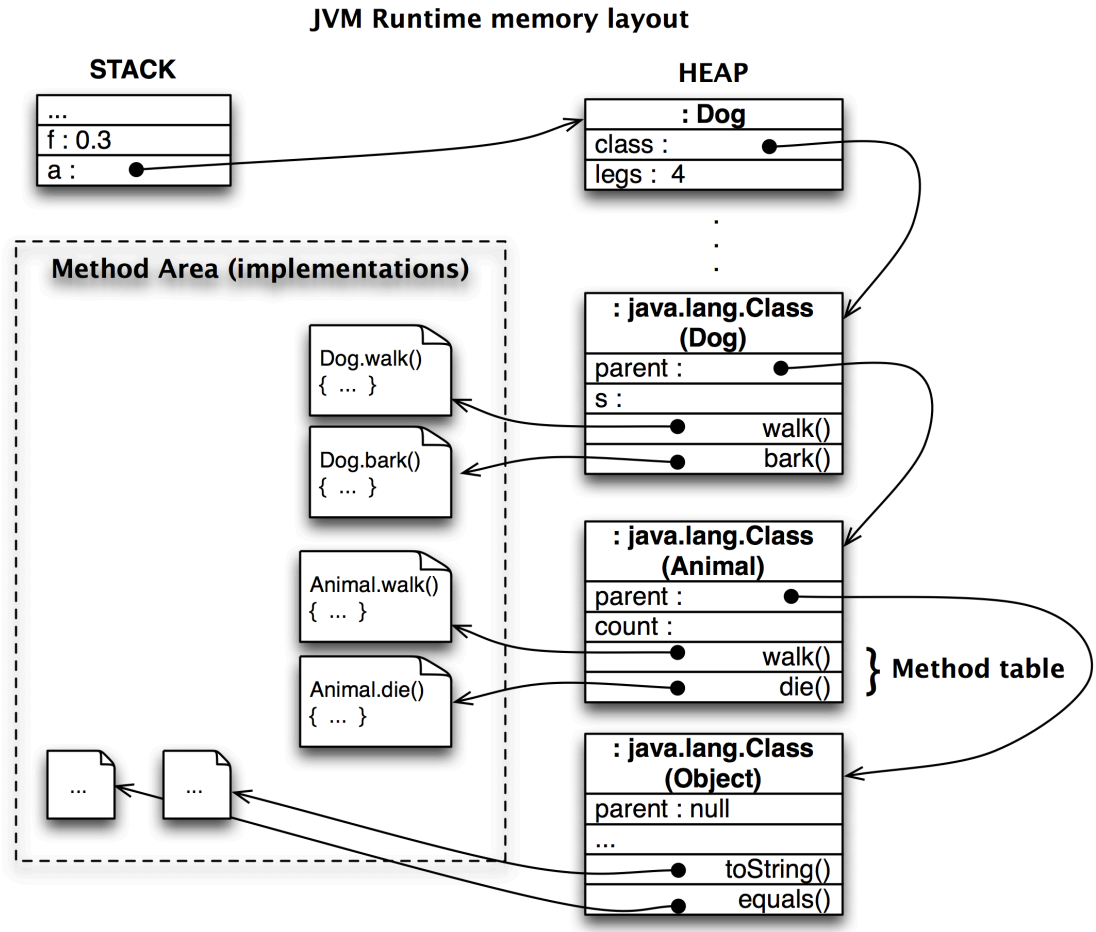
Tiempo de enlace

- Al momento en que se conoce el tipo de un dato (una variable, una expresión, etc.) se le conoce como **tiempo de enlace** y es un concepto asociado con el de tiempo de compilación y tiempo de ejecución.
- Tenemos, por tanto *early binding* y *late binding*, también llamados **enlaces estático** y **enlace dinámico**.
- Si para determinar el tipo de un dato se usa *enlace estático*. los errores que se produzcan los detectará y emitirá el compilador.
- Si para determinar el tipo de un dato se usa *enlace dinámico*, los errores que se produzcan en los tipos ya no los detectará el compilador, ¡incluso aunque sea un lenguaje compilado! Se trata de errores que se producen en tiempo de ejecución y, por tanto, están ligados a la idea de excepción que ya vimos. En Java, por ejemplo, los errores que se producen al usar incorrectamente enlace dinámico lanzan excepciones.
- Pero, si no es el compilador, ¿quién supervisa estos errores?
    1. En Java es la JVM que va ejecutando el binario y comprobando cosas a la vez.
    2. En JavaScript es el intérprete/motor el que va comprobando las cosas.
    3. En C++ es código objeto adicional generado por el compilador.
- Fijémonos que aunque Java soporta la idea de enlace dinámico para contextos de herencia, el compilador aún se encarga de ciertas comprobaciones; así en el ejemplo anterior no es posible hacer al final del código lo siguiente pero porque el compilador no lo permite, ya que 'meow()' no está definido a nivel de 'Animal':

```
a.meow();
```

Representación de las clases en la memoria de la JVM

- Seguimos con el código de *Animal* , *Cat* y *Dog*. Vamos a ver cómo aún hay contextos donde se usa enlace estático (por ejemplo, las llamadas a métodos estáticos) y cómo hace la JVM para saber qué método toca llamar cuando hacemos *a.walk()*.
- En la figura puedes ver un ejemplo de la organización en memoria de los objetos y clases del fragmento de programa del ejemplo en ejecución. Las variables locales y argumentos de un método se almacenan en la **pila (stack)**. Su espacio de memoria se libera cuando el método finaliza (por ello crece y decrece como una pila). Los objetos se almacenan en el **heap** o, como se conoce en la jerga de C++, la *memoria dinámica*. Cuando creamos un objeto lo que se crea en la pila es la referencia (es decir, una dirección de memoria) y en el heap se crea el objeto en sí. En el **área de métodos (method area)** se almacenan las implementaciones de los métodos (sus instrucciones).



- Los objetos de clase *java.lang.Class* los crea el *class loader* de la JVM cuando se crea por primera vez un objeto de una determinada clase. Puedes pensar en estos objetos como la representación en memoria de un archivo '.class', que contiene toda la información acerca de la estructura, interfaz e implementación de una clase. Se puede acceder a este objeto con *Animal.class* o *a.getClass()* (por cierto, este método viene de Object y lo tienen todos los objetos por herencia). Fíjate en que estos objetos guardan el espacio de memoria necesario para los atributos estáticos.
- El método `walk()` tiene enlace dinámico. En la llamada `a.walk()` la JVM recorre el diagrama, desde `a` en la pila, hasta llegar a la implementación `Dog.walk()`, que ejecutará. Lo mismo ocurre

con la llamada `a.die()`, sólo que esta vez *Dog* no ha sobrescrito este método, con lo cual la JVM seguirá buscando en su superclase una implemen tación de `die()`, encontrándola en *Animal*.

- La llamada a `Animal.getCount()` no tiene que recorrer el grafo siguiendo las flechas porque usa enlace estático. Ya se enlazó al compilar el código fuente a *bytecode*.
- El **recolector de basura (garbage collector, GC)** de la JVM entraría en acción a partir del momento en que desapareciera la única referencia que tenemos al objeto *Dog* en el heap, es decir, la variable `a` que está en la pila. El GC liberaría la memoria de este objeto, ya que nuestro programa ya no mantiene referencias a él y es por tanto inaccesible. Fíjate en que los objetos *java.lang.Class* se mantendrán en memoria siempre que existan referencias a ellos.
- Recuerda que en Java se usa siempre *enlace dinámico* para métodos de instancia. En C++ se usa *enlace estático* por defecto, salvo para métodos virtuales, donde se usa enlace dinámico.
- En la figura 5, puedes ver un ejemplo de código que representa la situación en memoria si implementamos un constructor de copia como copia superficial o copia profunda.
- Te recomendamos realizar estos Ejercicios sobre gestión de memoria y enlace dinámico que encontrarás en nuestra web.

Notas adicionales

- En el código en bytecode *this* es en los métodos de instancia un parámetro implícito y así se sabe siempre sobre qué objeto es sobre el que ha de actuar un método pese a que no hay una copia del código del método para cada objeto.
- Prueba a compilar un programa simple y desensamblarlo con la utilidad `javap`.
- Explora la diferencia en C++ entre manejar objetos mediante referencias, punteros, u objetos automáticos, representándolos gráficamente y usando un ejemplo similar al de los gatos y perros. Recuerda que una referencia o puntero en C++ es similar a las referencias Java, mientras que un objeto automático se crearía directamente en la pila, sin referencias:

```
Dog perro1; // objeto automático, se crea en la pila
Dog* perro2 = new Dog(); // se crea un puntero a Dog 'perro2' en la pila y apunta a un
objeto Dog que se crea en el heap
Dog& perro3 = *perro2; // se crea una referencia a Dog perro3 en la pila, que apunta al
mismo objeto del heap al que apunta perro2.
```

Enlaces de interés

Gestíon de memoria en Java (en inglés)

Java (JVM) Memory Model – Memory Management in Java