TeoriaTPP.pdf

Luneko

Tecnologias y Paradigmas de la Programacion

2º Grado en Ingeniería Informática del Software

Escuela de Ingeniería Informática
Universidad de Oviedo

Tema 1: Lenguajes y Paradigmas de la Programación

Lenguaje de Programación, compilador e intérprete

Lenguaje de Programación: **lenguaje artificial** para escribir instrucciones, algoritmos o funciones que pueden ser ejecutadas por un ordenador. Acerca el nivel de abstracción del humano al de una máquina.

Traductor: **programa** que procesa un texto fuente (origen) y genera un texto objeto (destino).

Compilador: **traductor** que transforma código fuente de alto nivel en código fuente de bajo nivel. (caso particular de traductor)

Intérprete: **ejecuta** las instrucciones de los programas.

Características de los lenguajes de programación

Nivel de abstracción

Mide en qué medida el lenguaje esta más próximo al **alto nivel** (humano) o al **bajo nivel** (ordenador).

El lenguaje ensamblador y código máquina son de bajo nivel.

'C' es considerado como el único lenguaje de medio nivel.

El resto de los lenguajes de programación son de alto nivel.

Los lenguajes específicos del dominio (DSLs) tienen un muy alto nivel de abstracción.

En función del dominio

Pueden ser de **propósito general** o **específicos** de un dominio (DSLs):

DSLs: lenguajes específicos de un dominio

SQL (bases de datos), Logo (dibujo), R (estadística)…

El uso de estos lenguajes ha aumentado en los últimos años por el Modelado Específico de Dominio (DSM) y Desarrollo Dirigido por Modelos (MDD)

Lenguajes de propósito general: Sirven para resolver cualquier problema computacional

Java, C++, C#..., Pascal…

Soporte de concurrencia

Permite la creación de programas mediante un conjunto de **procesos/hilos** que interactúan entre sí y que pueden ser ejecutados de forma **paralela**

Los programas concurrentes pueden ser ejecutados:

- En un único procesador: intercalando la ejecución de los procesos/hilos.
- En paralelo: cada proceso/hilo a procesadores distintos (misma máquina o distribuidos en una red de computadores)

Lenguajes concurrentes: Go, Ada, Erlang…

Muchos lenguajes no ofrecen la concurrencia como parte de ellos, pero sí por librerías estándar: Java, C++, C#...

No concurrentes: C y C++ 2003

En función de su implementación

Compilados o **interpretados** (no excluyente)

Java y C# son lenguajes compilados e interpretados.

Un compilador JIT **transforma el código** a interpretar en código nativo propio de la plataforma donde se ejecuta el intérprete (si es antes de su ejecución -> AoT (Ahead of Time)).

Lenguajes de naturaleza interpretada -> JavaScript, MathLab

Lenguajes de naturaleza compilada -> C, C++, Pascal

Lenguajes primero compilados y luego interpretados -> Java

Lenguajes diseñados para ser compilados JIT: Self, C#

Lenguajes tanto compiladas como interpretados: Haskell, Lisp

Sistema (comprobación) de tipos

Las comprobaciones de tipos pueden ser **estáticas** (compilación) o **dinámicas** (ejecución), no excluyente.

Comprobación estática: **detección temprana de errores**, mayor **rendimiento**.

Comprobación dinámica: mayor **adaptabilidad** y **metaprogramación dinámica**.

- Lenguajes de Scripting: dinámicos orientados a la unión de componentes. ActionScript, JavaScript.
- Lenguajes dinámicos: permiten desarrollar APPs finales utilizando solo ese lenguaje de programación. Python, JavaScript

Los lenguajes con solo comprobación estática son compilados.

No existen lenguajes interpretados **puros** con solo comprobación dinámica.

Existen lenguajes compilados con comprobación dinámica.

Los lenguajes con solo comprobación dinámica tienen fase de interpretación.

Representación del código fuente

Lenguajes visuales: representan entidades de su dominio mediante notación virtual en lugar de textual (UML, LabVIEW…). Existen números lenguajes visuales **específicos de su dominio**.

Lenguajes textuales: usan **ficheros de texto** para contener su código fuente (Java, C#...).

Paradigmas de programación

Paradigma: **aproximación** para construir programas caracterizado por la **abstracción** y conceptos para **representar** dichos **programas**.

- **Clasifica** lenguajes de programación
- Un lenguaje de programación puede seguir más de un paradigma (**multiparadigma**).
- No constituyen características de un lenguaje

Imperativo vs. Declarativo

Clasificación de los paradigmas (o incluso lenguajes).

Imperativo

Describe los programas en términos de **sentencias** que **cambian el estado del programa**.

- Se debe de especificar **cómo** se debe de realizar cada tarea.
- **Abstracción** más cercana al ordenador que los declarativos
- Lenguajes: C, Java, C#, Pascal…
- Paradigmas: estructurado basado en procedimientos, orientado a objetos…

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en ing.es

1/6
Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

Si ya tuviste sufi con tanto estudio...
Te dejamos este espacio
para desahogarte.

Pinta, arranca,
llora... tú decides ;)

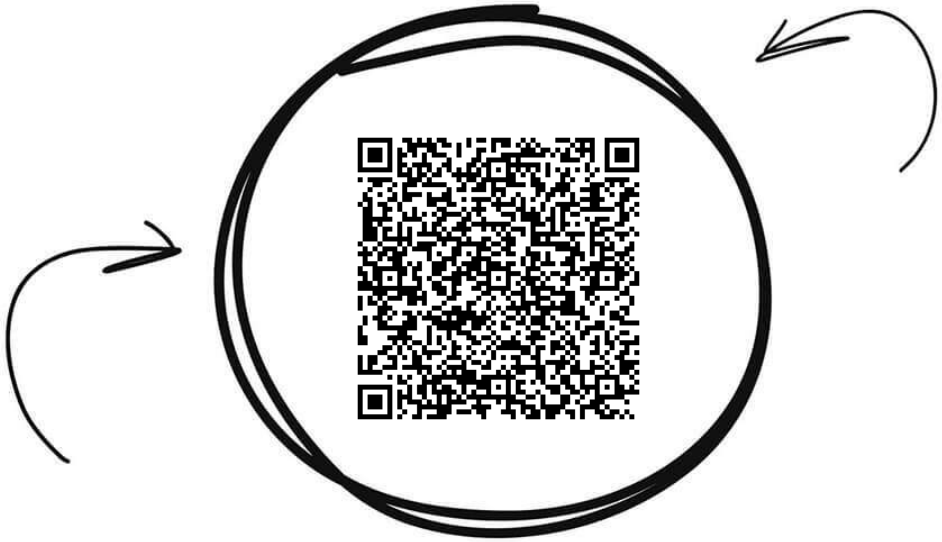
¿Te sientes más liberado?
Sigue siéndolo con la **Cuenta NoCuenta:**
libre de comisiones*, y de lloraditas.

¡Quiero una de esas!

*TIN 0 % y TAE 0 %.

do your thing

Tecnologias y Paradigmas de...



Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

Declarativo

Escribir programas que especifican **qué** se quiere obtener sin especificar cómo debe de hacerse.

- Se debe de declarar lo que queremos obtener
- Se utilizan las **abstracciones** del paradigma específico para representar el qué
- Lenguajes: Prolog, SQL…
- Paradigmas: lógico, funcional, basado en restricciones…

Principales paradigmas

Estructurado basado en procedimientos (imperativo)

Formado por la unión del paradigma estructurado y del procedural.

P. estructurado: utiliza **3 estructuras de control** (secuencial, condicional e iterativo) que se pueden agrupar en procedimientos o subrutinas.

- Se evitan instrucciones de **salto** (condicionales e incondicionales)
- Aumenta **legibilidad** y mantenibilidad
- Las estructuras de control se pueden **anidar.**

Define el procedimiento como **primer mecanismo de descomposición** (un procedimiento contiene una lista ordenada de instrucciones)

Incorpora la **programación estructurada** (por ello constituyen un solo paradigma)

Previene el acceso indebido a variables (**ámbito**)

Función (procedimiento que devuelve un valor):

- Puede generar efectos colaterales
- No ofrecen funciones de orden superior
- No implementan cláusulas

Lenguajes: C, Pascal…

Orientado a objetos (imperativo)

Utiliza los **objetos**, unión de **datos y métodos** como principal método de abstracción.

Se basa en la idea de modelar **objetos reales** mediante la codificación de **objetos software**.

Programa: conjunto de objetos interactuando

Elementos propios del paradigma: encapsulamiento, herencia, polimorfismo y enlace dinámico.

Dos modelos:

- Basado en clases: todo objeto tiene que ser **instanciado** de una clase que define la estructura y comportamiento del objeto. C++, Java, C#
- Basados en prototipos: los objetos son la única entidad. Se define un **objeto prototipo** y el resto de los objetos se **clonan**. JavaScript, Python

Se dice que un **lenguaje OO** es **puro** cuando todas las abstracciones ofrecidas son objetos.

Paradigma funcional (declarativo)

Basado en la utilización de funciones que manejan **datos inmutables**. Éstas devuelven el dato modificado sin machacar el original.

Un programa es un **conjunto de funciones invocándose entre sí.**

Hace uso de la **recursividad** en lugar de la iteración.

Los **lenguajes funcionales puros** no utilizan la asignación ni secuenciación de instrucciones.

- Lenguajes funcionales: Lisp, Erlang

- Lenguajes funcionales puros: Miranda, Clean

Paradigma lógico (declarativo)

Basado en la programación de ordenadores mediante **lógica matemática.** El programador describe un conocimiento por reglas **lógicas** y axiomas.

Un **demostrador de teoremas** con **razonamiento hacia atrás** resuelve problemas a partir de consultas.

- Lenguaje: Prolog

Otros paradigmas

Orientación a aspectos (imperativo)

Modularizan la funcionalidad de una aplicación que se encuentra **entremezclada y dispersa** a lo largo del código.

Ejemplo de aspectos: persistencia, seguridad, logging, tracing, …

Las herramientas que dan soporte a aspectos se basan en técnicas de **instrumentación** o procesamiento (**tejido**) de **programas**.

El incremento de modularidad ofrece **beneficios** de mantenibilidad, legibilidad…

AspectJ: herramienta/lenguaje más utilizado para este paradigma.

AOSD: disciplina que sigue la separación de aspectos en todas las fases del ciclo de desarrollo de software.

Basados en restricciones (declarativo)

Las relaciones entre variables son expresadas mediante **ecuaciones** (restricciones). Éstas se expresan sobre distintos **dominios**: booleano, entero, racional…

Se implementan como:

- **Lenguaje** propiamente dicho: eclipse
- **Ampliación de otro lenguaje**: CLP (R), B-Prolog
- **API** de un lenguaje imperativo: Disolver

Tiempo real

Tiene que llevar a cabo **computaciones en un tiempo máximo dado** (independientemente de la carga del sistema).

Hard real-time: si no realiza la operación en el tiempo establecido produce un **fallo crítico** en el sistema.

Soft real-time: si no realiza la operación en el tiempo establecido deja de realizar otras tareas para **recuperarse del fallo** del sistema.

Lenguajes: Ada, Real-Time Java

Guiado por eventos

Consiste en un bucle principal que escucha **eventos** (acciones de usuario, datos de sensores o mensajes enviados por otros programas o hilos) y con código en forma de **callback** que se ejecuta en función del evento detectado.

Se pueden escribir programas con este paradigma en casi todos los lenguajes de programación.

Programación basada en autómatas

Los programas son una **máquina finita de estados** o cualquier **autómata formal**

- Son un **ciclo de pasos definidos por el autómata**
- El tiempo de ejecución está separado en los pasos del autómata
- Cada paso de ejecución puede dividirse en subpasos que se ejecutan dependiendo del estado actual.

- Cualquier comunicación entre pasos solo es posible mediante un conjunto de variables especiales (estados).
- El estado del programa completo depende solamente de estas variables de estado

Usos

- **Análisis léxico, sintáctico y semántico** de lenguajes de programación.
- Necesario en la programación guiada por eventos para el paralelismo.
- La máquina finita de estados y estados se usan en la **especificación**
- UML usa diagramas de estado para describir el comportamiento.

Programación reactiva (declarativo)

Usa los conceptos de **flujos de datos** y **la propagación del cambio.**

Permite expresar flujos de datos estáticos (arrays) o dinámicos (eventos) y mostrar que existe una dependencia entre ellos.

Patrón de diseño **Observer**: un objeto Publisher (implementa IObservable<T>) se comunica con otros Subscribers (implementa IObserver<T>) enviándoles (push) o dejándoles disponibles (pull) mensajes hacia o para ellos.

ReactiveX: **API más popular** para programas que usen programación reactiva.

Se basa en los patrones de diseño Observable e Iterator y en la **Programación Funcional**

Tecnología de la programación

Elementos y técnicas ofrecidas para los **distintos paradigmas y lenguajes** de programación para diseñas, construir y mantener aplicaciones de forma correcta, robusta, segura y eficiente.

C#

Elementos del **paradigma OO**: encapsulamiento, herencia, polimorfismo, genericidad, autoboxing

Elementos del **paradigma funcional**: funciones lambda, funciones de orden superior, cláusulas, continuaciones

API estándar de programación **concurrente y paralelización**

Elementos incluidos en las nuevas **tendencias de lenguajes de programación**: reflexión, metaprogramación, generación dinámica de código, programación orientada a atributos, tipado estático y dinámico, lenguaje integrado de consulta.

Posee implementaciones para Windows, Linux y Mac OS

Estándar **ECMA e ISO**

Lenguaje muy utilizado a nivel profesional con tendencia ascendente

Tema 2: Paradigma Orientado a Objetos

Paradigma Orientado a Objetos

Utiliza los **objetos**, unión de datos y métodos como principal abstracción formando **programas** como interacciones entre objetos.

Se trata de modelar **objetos reales** mediante la codificación de **objetos software**.

Un programa está constituido por un conjunto de objetos que intercambian mensajes entre sí (interactúan).

Abstracción

Expresa las características esenciales de un objeto que lo distingue de los demás. Es el principal método de abstracción de los lenguajes de programación para representar sus abstracciones son sus tipos.

Encapsulamiento

Proceso de almacenar en un **mismo compartimento** elementos de una abstracción que constituyen su estructura y comportamiento.

- Los objetos encapsulan en una misma entidad datos y métodos.

La **ocultación de información** permite distinguir entre que partes de la abstracción están disponibles al resto de la aplicación (diversos niveles de ocultación)

Cada objeto está aislado y expone su interfaz a otros objetos que especifica como pueden interactuar.

Beneficios

- Ofrece **mantenibilidad.**
- Permite la **reutilización** de código (interfaz clara)
- Ofrece **robustez** (para manipular la estructura se utilizan operaciones bien definidas que evitan errores).

Propiedades

Para acceder al estado de los objetos aprovechando los beneficios del **encapsulamiento** y de la **mantenibilidad**.

Es decir, oculta el estado interno ofreciendo un acceso indirecto mediante las propiedades. Éstas pueden ser de **lectura y/o de escritura**, tener cualquier nivel de ocultación, ser estáticas, abstractas y sobrescribirse.

Modularidad

Subdividir una aplicación en partes más pequeñas (módulos) siendo cada una de ellas tan independientes como sea posible.

Cada **módulo** debe de poder ser compilado por separado para utilizarlo en diversos programas (reutilización).

Un módulo puede estar formado por:

- Funciones/métodos
- Clases/tipos
- Espacio de nombres/paquetes
- Componentes

Acoplamiento y cohesión

Acoplamiento: nivel de **independencia** entre módulos.

Cohesión: nivel de **uniformidad y relación** que existe entre las distintas responsabilidades de un módulo.

Un **bajo acoplamiento y alta cohesión** proporciona reutilización y mantenibilidad del código.

Sobrecarga

De métodos

Permite dar distintas implementaciones a un mismo identificador de método.

Para ello cada método debe diferir de los otros en:

- Nº de parámetros
- Tipo de parámetros
- Paso de algún parámetro (valor, **ref** u **out**)

De operadores

Permite modificar la semántica de los operadores del lenguaje.

C#: ++, --, [ ], cast y conversiones implícitas (aunque apenas se usan)

Herencia y polimorfismo

Herencia

Mecanismo de **reutilización de código** por el cual el **estado** de una instancia derivada está definido por la unión de las estructuras de la clase base y la derivada. El conjunto de mensajes (**interfaz**) que acepta un objeto derivado es la unión (herencia) de los mensajes de su clase base y derivada.

Polimorfismo

Mecanismo de **generalización** que hace que la abstracción más general pueda representar abstracciones más específicas.

- El tipo general representa varias formas (poli + morfismo).

Las **referencias derivadas** promocionan a **referencias base** (subtipado)

Con referencias polimórficas solo se podrán pasar mensajes del tipo de la referencia.

Para conocer el tipo dinámico de un objeto hay que hacer cast con los operadores **is/as** (InvalidCastException)

Enlace dinámico

Los métodos derivados se pueden **especializar** en las clases derivadas.

Si queremos que se llame al método implementado por el objeto, debemos de hacer uso del **enlace dinámico:** en tiempo de ejecución se invoca al método del tipo dinámico implementado por el propio objeto (no al estático declarado en la clase base).

Enlace dinámico en Java:

- Lo permite **por defecto**, siendo las clases hijo capaces de llamar a un método, redefinirlo y cambiar su comportamiento
- Se puede prohibir declarando el método/clase como **final**

Enlace dinámico por autorización (C#):

- Los métodos redefinibles deben marcarse como **virtual**, si no, métodos con misma signatura en clases hijas deben marcarse como **new**.
- Aunque el padre sea virtual, los hijos deben marcarse como **override**
- Las palabras reservadas se pueden aplicar a métodos y las propiedades.

Clases abstractas e Interfaces

Clases y métodos abstractos

Cuando en una abstracción necesitamos que un **mensaje** forme parte de la interfaz, pero no podemos implementarlo, se declara como abstracto.

- El método no se implementa (es un mensaje)
- Utilizar la palabra reservada **abstract.**

Los métodos abstractos deben de ser redefinidos (**override**) ofreciendo así enlace dinámico.

Toda clase con al menos un método abstracto debe de ser una **clase abstracta** (no hace falta que tenga ninguno para serlo)

Herencia múltiple

Una clase hereda, **directamente**, de **más de una clase**.

Conflictos:

- Coincidencia de nombre: se hereda de más de una clase con un miembro de igual identificador. Genera ambigüedad.
- Herencia repetida: se hereda más de una vez de una clase por distintos caminos (duplicidad de miembros).

Interfaces

Usos:

- Casi no se utilizaba herencia múltiple de implementación, se sustituyó por composición
- Normalmente se buscaba polimorfismo, se incluyó al lenguaje el concepto de interfaz

Necesitamos polimorfismo múltiple pero no se puede entonces utilizamos una interfaz (conjunto de mensajes públicos que ofrecen un conjunto de clases)

Este concepto se considera un **tipo**:

- Interfaz -> **polimorfismo múltiple**.
- Clase o interfaz -> puede derivar de más de una interfaz.

Excepciones

El compilador no es capaz de detectar todos los errores de un programa. Existen errores que pueden producirse en función del contexto (en tiempo de ejecución). Por ello, se debe de dotar al lenguaje de mecanismos de control dinámico de errores.

Objetivo del manejo de excepciones

Los errores en tiempo de ejecución no deberían de evitar el desarrollo de software:

- Reutilizable: el manejo de errores puede ser diferente para cada situación.
- Robusto: el sistema deberá obligar al programador a gestionar los errores de forma diferente al flujo normal de ejecución.
- Extensible: en función de la abstracción un error puede convertirse en otro error o manejarse.

Excepción

Se trata de un **evento** que se lanza en un momento de ejecución y que impide que la ejecución prosiga por su flujo normal.

El manejo de excepciones separa:

- La abstracción que detecta el error y lanza la excepción.
- Las distintas abstracciones que manejan la excepción del modo que más interese.

En C# todas las excepciones son **unchecked** y no se especifican si son lanzadas por un método (throws en Java).

Asertos

Se encuentran en la clase **Debug**, y están habilitados en el modo Debug y no en Release

La técnica más utilizada para implementarlos está basada en **compilación condicional**

(#if DEBUG #else #endif)

Condiciones que se han de cumplir en la correcta ejecución de un programa. Si se producen, se detiene la ejecución.

- No se deben de utilizar para detectar errores en tiempo de ejecución (no son reutilizables ni extensibles).
- Detectan situaciones que nunca deberían de ocurrir.
- Detectan errores de programación.

Genericidad

Propiedad que permite construir abstracciones modelo para construir otras asbtracciones.

Beneficios:

- Mayor **robustez** (detecta más errores en tiempo de compilación).
- Mayor **rendimiento**.

Genericidad acotada

Método -> public/private T Método<T>(T param) { … }

Clase -> class clase<T> { … }

Por omisión, los elementos genéricos son objects.

La **genericidad acotada** permite hacer que los distintos tipos genéricos sean más específicos.

- Limitan su genericidad
- Se permite un mayor paso de mensajes

Utilizaremos colecciones genéricas como: IEnumerable<T>, IList<T>, List<T>…

Genericidad en C#

Colecciones genéricas:

- IEnumerable <T>
- IList <T> y List <T>
- IDictionary <TKey, TValue> y Dictionary <TKey, TValue>

Genericidad en Java

Tiene varias limitaciones.

Inferencia de tipos

Capacidad para deducir automáticamente el tipo de una excepción. Cuanta menos información de tipos tenga el programador, más **avanzada** será la inferencia de tipos.

(F#) let f a b = a + b + 100
Infiere el tipo de la función f a -> int f (int a, int b)

3 escenarios:

- Métodos genéricos
- Variables locales declaradas implícitamente (**var**)
- Funciones lambda

Inferencia en métodos genéricos

Permite no especificar el tipo de los métodos en su invocación.

```
static void Swap(ref T lhs, ref T rhs) {
        T temp; temp = lhs;
        lhs = rhs; rhs = temp;
}
static void Main() {
        int a = 1, b = 2;
        Swap(ref a, ref b);
        double c = 3.3, d=4.4;
        Swap(ref c, ref d);
        Swap(ref a, ref d); // Compiler Error
}
```

Variables declaradas implícitamente

Permite no declarar el tipo de las variables locales utilizando la palabra reservada **var** y asignando una expresión a su declaración.

Útil para:

- Tipos con nombres largos
- No se puede identificar el tipo
- No existe un tipo explicito

Tema 3: Fundamentos del paradigma funcional

Paradigma **declarativo** basado en la utilización de **funciones** que manejan datos inmutables. En lugar de cambiar el dato, se llama a una función que devuelve el dato modificado sin machacar el original.

Las funciones no generan efectos colaterales: el valor de una expresión depende únicamente de sus parámetros.

Calculo lambda

Es el origen del paradigma funcional. Se trata de un sistema basado en la **definición de funciones** (abstracción) y su **aplicación** (invocación).

Es considerado como **el lenguaje más pequeño universal de computación**.

Actualmente se utiliza para describir nuevos elementos de programación.

Expresiones lambda

Se define como

- Una abstracción lambda λx. M (M, N, M1, M2…) donde 'x' es una 'var' y 'M' es una expresión lambda.
- Una aplicación MN donde 'M' y 'N' son expresiones lambda.

Aplicación, Reducción – β

La aplicación de una función representa su invocación y se define del siguiente modo:

Siendo x una variable y M, N expresiones lambda,

(λx. M) N → M[x := N] (o M[N/x])

Lo de la derecha de la implicación representa M donde todas las apariciones de x son sustituidas por N, esta sustitución se llama **REDUCCIÓN–B.**

Teorema de Church – Rosser

En algunos términos se pueden aplicar múltiples reducciones – β y el orden en el que se hagan no afecta al resultado final.

Variables libres y ligadas

En una abstracción λx. xy

- X: variable ligada
- Y: variable libre

En una sustitución, solo se sustituyen las variables libres. En ocasiones, para evitar conflictos entre las variables de distintas expresiones, se creó la conversión – α. De tal forma que todas las operaciones de una variable ligada en una misma abstracción se pueden renombrar a una nueva variable.

El problema de la parada

Dada la especificación de un programa, se trata de demostrar si finalizará su ejecución o no.

Es imposible describir un algoritmo que resuelva el problema de la parada para todos los posibles programas t entradas (problema no decidible).

- Hay lenguajes que permiten asegurar que un programa termina.
- En calculo lambda se pueden escribir programas que no terminan.

Lógica como base del Software

LÓGICA: ciencia que se dedica al estudio de las formas válidas de inferencia y su demostración.

DEMOSTRACIÓN: prueba convincente de que un teorema es necesariamente cierto. Suele realizarse mediante razonamiento deductivo.

EVIDENCIAS: distintas demostraciones de que una proposición es cierta.

Isomorfismo de Curry – Howard

Establece una relación directa entre los programas software y demostraciones matemáticas.

- Existe una correspondencia entre tipos y proporciones.

Lógica	Equivale al	Tipo
⊃ (→)		Función
∧		Tipo producto (Tupla o Registro)
∨		Tipo suma (Unión)
true		Tipo Top (Object)
false		Tipo Bottom (void)
∀		Genericidad

- Existe una correspondencia entre programas y evidencias que demuestran la proposición descrita por su tipo.

Demostración de propiedades

La lógica se puede utilizar como mecanismo formal para la demostración de propiedades del software.

El mejor ejemplo es la **VERIFICACIÓN DE PROGRAMAS**, que consiste en demostrar que un programa es correcto conforme a una especificación.

- Por ejemplo un algoritmo de ordenación será correcto cuando se demuestre que tras su invocación todos los elementos están y ordenados para cualquier entrada.

Lógica y Paradigma Funcional

El paradigma funcional es el más utilizado para realizar demostraciones sobre programas porque:

1. Toda computación se puede expresar en cálculo lambda.
2. Hay una traducción directa con la lógica.

Existen asistentes de demostradores que permiten demostrar propiedades de programas.

- Se basan en lenguajes funcionales.
- Realizan demostraciones mediante deducción natural.
- Permite la extracción de programas.

Funciones, Entidades de Primer Orden

En el paradigma funcional se identifican a las funciones como **entidades de primer orden**: **funciones de 1ª clase.** Esto quiere decir que las funciones son consideradas como un tipo más.

Funciones de orden superior

Una función es de orden superior si **recibe** como parámetro **o retorna** una función.

La función doble aplicación es una función de orden superior: λf. (λx. f(fx))

Delegados

Las funciones pueden ser consideradas entidades de primer orden gracias a los delegados. Representa un método de instancia o de clase (static). Las variables de tipo delegado representan un modo de referenciar un método.

- El paso de estas variables implica el paso de métodos a otros métodos.

- La asignación permite parametrizar estructuras de datos y algoritmos en base a otras funcionalidades.

Patrón Observer

Los suscriptores (Observers) se registran y desregistran en un asunto (Subject). Cuando sucede un evento, el asunto (Subject) notifica de tal evento a los elementos suscritos (Observers).

Una instancia de un delegado puede coleccionar un conjunto de métodos (+=,-=,=). Cuando se invoca a un delegado, se invocará a todos los suscriptores registrados en ese delegado.

Tipos de delegados predefinidos

- **FUNC**: Siempre retorna un valor
- **ACTION**: Nunca retorna un valor
- **PREDICATE**: Siempre retorna un bool

Delegados anónimos

Sintaxis para definir una variable delegado indicando sus parámetros y su cuerpo, lo que permite escribir la función en el momento de pasarla.

Expresión lambda

Permiten escribir el cuerpo de funciones completas como expresiones (mejora sobre los delegados).

- Se deben de especificar los parámetros separados por comas.
  - Se pueden anteponer los tipos
  - Si la función solo recibe uno, se pueden omitir los paréntesis,
  - Si no recibe parámetros debemos de indicarlo con ().
- **=>** separa los parámetros del cuerpo
- Si el cuerpo es una única sentencia, no es necesario escribir return ni llaves

Los tipos de expresiones lambda promocionan a los tipos de delegados predefinidos.

Bucles y Recursividad

La programación funcional pura no posee el concepto de iteración (bucle). Hacen uso de la recursividad.

En cálculo lambda las funciones no poseen nombres. Por lo tanto, las funciones recursivas deben de implementarse como funciones de orden superior pasándose a sí mismas como parámetro.

λx. if x=0 or x=1 then 1 else x*?(x-1)

Combinador de punto fijo

Función de orden superior que cumple: fix f → f (fix f).

Aplicar fix a f(fix f) se retorna f pasando una nueva invocación a fix f como primer parámetro.

- Recordemos...

a) `fact ≡ λf.λx.if x=0 or x=1 then 1 else x*f(x-1)`
b) `fix fact → fact (fix fact)`

- Evaluemos pues **`fix fact 3`** (las expresiones evaluadas aparecen subrayadas)

```
1. fix fact 3                    b)→
2. fact (fix fact) 3             a)→
3. 3 * (fix fact) 2              b)→
4. 3 * fact (fix fact) 2         a)→
5. 3 * 2 * (fix fact) 1          b)→
6. 3 * 2 * fact (fix fact) 1     a)→
7. 3 * 2 * 1
```

COMBINADOR Y: Una de las funciones fix más conocidas

fix ≡ λf.(λx. f(xx))(λx. f(xx))

$$\boxed{fix\ fact} \equiv$$
$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))\ fact \rightarrow$$
$$\boxed{(\lambda x.fact(xx))(\lambda x.fact(xx))} \equiv_\alpha$$
$$(\lambda y.fact(yy))(\lambda x.fact(xx)) \rightarrow$$
$$fact\ (\ \boxed{(\lambda x.fact(xx))(\lambda x.fact(xx))}\ )\ \equiv$$
$$fact\ (fix\ fact)$$

Clausura

Función de primer orden junto con su ámbito:
tabla que guarda las referencias a sus variables libres.

En cálculo lambda λy. …x es una clausura.

- (λx. …x…) (λy. … x) -> la variable 'x' es libre, pero guarda referencia a lo anterior.

Las variables libres de una clausura representan el **estado** que puede estar oculto cuando el ámbito de la variable finaliza.

Las clausuras pueden representar:

- **Objetos** u ocultación de información.
- **Estructuras de control.**

Currificación

Técnica para transformar una función de varios parámetros **en una función con solo uno.**

- La función recibe un solo parámetro y puede devolver otra función que se pueda llamar con el 2º parámetro.
- La invocación se convierte en llamadas a funciones encadenadas.
- F(1)(2)(3) vs. F(1,2,3)

Principal beneficio**: la aplicación parcial.**

Aplicación parcial

Con funciones currificadas es posible realizar su invocación parcial. Existen lenguajes en los que todas sus funciones son currificadas.

Aplicación parcial: pasar un número menor de parámetros en la invocación de la función de forma que el resultado es otra función con un número menor de parámetros.

En C#

- Las funciones no están currificadas por omisión.
- Los operadores no son funciones.

La aplicación parcial se puede utilizar para **escribir menos código**. En lugar de crear múltiples elementos con una parte común, definimos dicha parte una vez y la reutilizamos:

- Definimos la parte común
- Guardar en un delegado (parcialmente-aplicada).
- Invocar la función parcialmente-aplicada con el resto de los parámetros

Continuaciones

Representa el **estado de computación en un momento de ejecución**. Y se suele componer de:

- Estado de la pila.
- Siguiente instrucción por ejecutar.

Los lenguajes que ofrecen continuaciones son capaces de almacenar su estado de ejecución y recuperarlo posteriormente.

FUNCIÓN CALL/CC: obtener el estado de computación.

Generadores

Se trata de una función que **simula la devolución de una colección de elementos** sin construir toda la colección (- memoria, el primer elemento es inmediato y solo generamos lo que usamos) y devolviendo un elemento cada vez que la función es invocada.

Una implementación de los generadores es mediante continuaciones.

En C# se implementan generadores mediante yield.

Evaluación perezosa

Técnica por la que **se demora la evaluación de una expresión hasta que se utiliza**. Lo contrario sería la evaluación ansiosa o estricta.

Beneficios:

- Menor consumo de memoria.
- Mayor rendimiento.
- Estructuras de datos potencialmente infinitas.

C# ofrece evaluación perezosa en las continuaciones implementadas en los generadores. Como la generación de elementos es perezosa, podemos generar colecciones de un nº infinito de elementos con yield. Para hacer uso de colecciones:

- **Skip**: desoír un conjunto de elementos de una secuencia, retornando los restantes.
- **Take**: para devolver un nº concreto de elementos contiguos desde el principio.

Transparencia referencial

Una expresión es referencialmente transparente si **se puede sustituir por su valor sin que cambie la semántica** del programa (lo contrario a opacidad referencial).

Los lenguajes que ofrecen transparencia referencial son considerados como **lenguajes funcionales puros.**

Elementos que impiden la transparencia referencial:

- **Variables globales mutables.**
- **Asignación destructiva** (+=, -=, *= …).
    - En estos dos primeros casos la evaluación de una variable depende de sus asignaciones previas
- **Funciones impuras** (Random, E/S).
    - Una función es pura si siempre devuelve el mismo valor ante los mismos parámetros y no genera efectos secundarios (actualizar variables globales dispositivos e/s…)

La transparencia referencial permite aplicar razonamiento matemático a los programas para demostrar su corrección o razonar sobre su comportamiento. También se pueden realizar transformaciones en los programas para su simplificación, optimización o paralelización.

Memorización

Técnica de optimización que puede ser aplicada sobre expresiones con transparencia referencial.

La primera vez que se invoca una expresión, se almacena el valor en la caché y como son funciones que siempre van a devolver lo mismo, en la siguiente consulta se devolverá él valor almacenado en la caché.

Evaluación perezosa

Con el paso de parámetros perezoso se demora la evaluación de una expresión hasta que se vaya a usar. Esto se puede conseguir haciendo que los parámetros sean funciones y memorizando su evaluación.

Pattern Matching

Acto de comprobar si un conjunto de elementos sigue algún patrón determinado.

Elementos:

- Tipos
- Listas

- Strings
- Arrays

Patrones:

- Secuencias (expresiones regulares)
- Estructura de árbol (lista bidimensional)

En C#

Introspección: única forma por la que se puede ofrecer comprobación de patrones (operador is o GetType -> devuelve un String que representa al elemento).

La introspección en este escenario es costosa en rendimiento en tiempo de ejecución y no detecta errores en tiempo de compilación.

En F#

Se trata de una implementación del lenguaje ML. Éste hace uso exhaustivo de pattern matching. Hace uso exhaustivo de pattern matching.

Es posible poner condiciones a los patrones con when y el caracter comodín _.

```
type Figura =
| Circulo of double
| Rectangulo of double * double
| Cuadrado of double
| Triangulo of double * double
;;
let regular figura =
    match figura with
    // _ es un comodin: implica que puede valer cualquier cosa
    | Cuadrado (_) -> true
    // cierto si el ancho y el alto son iguales
    | Rectangulo(ancho, alto) when ancho = alto -> true
    // falso para el resto de rectángulos
    | Rectangulo(_, _) -> false
   // cierto si es isósceles
    | Triangulo(labase, altura) -> altura =
                      labase*System.Math.Sqrt(3.0)/2.0
    | _ -> false // falso para el resto de figuras (el circulo)
    ;;
```

. Se utiliza la palabra reservada when y el carácter comodín _.

F# también ofrece pattern matching de listas con el operador ::Utiliza el operador ::.

El patrón head::tail indica que:

- **head**: primer elemento de la lista
- **tail**: lista resultante de quitar el primer elemento.

Funciones de orden superior típicas

Filter: aplica un predicado a todos los elementos de una colección, devolviendo otra colección con aquellos elementos que cumplen el predicado.

Map: aplica una función a todos los elementos de una colección, devolviendo una nueva colección con los resultados obtenidos.

Reduce(Fold, Acc, Compress/inject): se aplica una funciones a todos los elementos de una lista, dado un orden y devolviendo un valor.

En C#

Se añadieron funciones de orden superior genéricas en System.Linq.Ling.

- Reciben colecciones IEnumerable utilizando métodos extensores.
- Reciben funciones de orden superior

Nombres de las funciones de orden superior:

- Map -> Select
- Filter -> Where
- Reduce -> Aggregate

Resto de funciones en diapositiva 91 del tema 3

Listas por comprensión

Permite crear listas basándose en las listas existentes. Usa la notación de creación de conjuntos de la teoría de conjuntos.

LINQ

Muy parecido a las listas por compresión. Empleando funciones Where y Select se pueden crear listas a partir de otras.

Tema 4. Programación concurrente y paralela

Introducción

Ley de Moore: el número de transistores por unidad de superficie en circuitos integrados se duplica cada 24 meses, sin encarecer su precio.

Según Moore, este crecimiento exponencial dejaría de cumplirse entre 2017 y 2022.

Arquitecturas multinúcleo

- Actualmente se amplía el número de núcleos en lugar de la frecuencia.
- Los microprocesadores multinúcleo incluyen + de 1 procesador en una misma CPU
- Ofrecen computación paralela de procesos e hilos
- Memoria compartida

Memoria distribuida

- Cada procesador posee una memoria privada
- Si un procesador necesita datos de otro, se comunican por un **canal** (Cluster, Cloud, Grid…)

Programación concurrente y paralela

Concurrencia: Varias tareas se ejecutan simultáneamente pudiendo interactuar entre sí

Las tareas (hilos o procesos) se pueden ejecutar en varios núcleos, procesadores o en un único procesador de forma simulada.

Enfatiza en la interacción entre tareas.

Paralelismo: Caso particular de la concurrencia en el que las tareas se ejecutan de forma paralela, **la simultaneidad es real**, **siempre**.

Enfatiza en la división de un problema en partes.

Procesos e hilos

Proceso: Programa en ejecución

- Formado por instrucciones, estado de ejecución y valores de los datos en ejecución
- Tareas concurrentes en diferentes procesadores

Se identifican por un **PID**.

Son representados por la clase **Process** (**System.Diagnostics**)

Hilo: Un proceso puede constar de varios hilos. Se trata de una **tarea de un proceso** que puede ejecutarse **concurrentemente**, compartiendo la memoria del proceso con el resto de hilos

- Cada hilo tiene PC, pila y valor de los registros.
- En procesadores multinúcleo, los hilos pueden ser tareas paralelas de un proceso

Son representados por la clase **Thread (System.Threading)**

Paralelización de algoritmos

- **Paralelización de tareas:** Tareas distintas que se pueden hacer a la vez
- **Paralelización de datos:** Misma tarea en porciones de los mismos datos

Pipeline

Modelo híbrido de los dos anteriores, sincronizamos la salida de una tarea como la entrada de la siguiente.

Paso asíncrono de mensajes

Primera aproximación para crear programas paralelos, cada mensaje asíncrono crea un nuevo hilo. En C# obtenemos esta funcionalidad se obtiene mediante **delegados y tasks**

async y await

Se apoya en el uso de Task y Task<T>

AWAIT: Se aplica sobre una expresión sobre la que se puede esperar

AYSYNC: Se aplica a métodos que usan await en su cuerpo, si dicho método ya ha terminado, continúa la ejecución, si no, se duerme hasta que termine

```
public static async Task AsynchronousTask() {
//Do something
await Task.Run(() => Thread.Sleep(2000));
Console.WriteLine(AsynchronousTask body);
return;
}
static void Main() {
...
var t = AsynchronousTask();
//This simulates more computations
Thread.Sleep(100);
Console.WriteLine(AsynchronousTask has ended);
t.Wait(); //Synchronization mechanism
```

Creación explícita de hilos

Haciendo uso del POO, la clase Thread encapsula un hilo de ejecución

THREAD.JOIN: El hilo que realiza la llamada se bloque hasta que finaliza el hilo que recibió el mensaje.

Condición de carrera

Se dice que múltiples tareas están en condición de carrera cuando su resultado **depende del orden de las tareas**. Debemos evitarlo, puesto que son un foco de errores.

Parámetros

- Usamos el POO para **encapsular** los datos de un hilo en un objeto
- Podemos usar una aproximación más funcional, también podemos pasar **parámetros a hilos**, en el hilo.Start(param).

Variables libres

Si se usan funciones lambda hay que tener cuidado con sus variables libres, puesto que cada hilo posee una copia de la pila de ejecución a partir del ámbito en el que se creó. Las variables ya declaradas serán compartidas por todos los hilos. Alternativas:

- **PASO DE PARÁMETROS** (preferible)
- **COPIA DE VARIABLES**

```
int local = 1;
int copia = local;
Thread hilo1 = new Thread( () => {
Console.WriteLine(Copia {0}., copia);
});
local = 2; hilo1.Start();
```

Se imprime Copia 1

Excepciones asíncronas

El try catch se debe poner en el código asíncrono, si se pone en código síncrono y un hilo que hemos iniciado lanza excepción, el catch es inútil, la excepción no se captura y el programa termina.

Context Switch

El cambio de contexto requiere **tiempo de computación** para almacenar y restaurar contextos y **memoria adicional** para guardar los distintos contextos.

Un número muy elevado de tareas puede conllevar caída de rendimiento.

Thread pooling

La creación y destrucción de hilos también conlleva un coste computacional y de memoria

Se debe:

- **Limitar el número máximo de hilos** de un proceso en relación con el número de procesadores y otros recursos de memoria
- Minimizar el número de hilos creados -> **reutilizar**

El CLI ofrece un mecanismo de Thread Pooling que optimiza el número de hilos concurrentes por procesador, eliminando los hilos cuando un proceso lleva tiempo sin crear nuevos hilos.

Foreground & Background Threads

Los hilos **foreground** son aquellos con los que la aplicación no terminará hasta que acabe la ejecución de todos los hilos foreground creados.

Un hilo **background** o **daemon** es aquel que será terminado cuando no queden hilos foreground

- Proveedores de servicios
- No confundir con workers

```
hilo.isBackground = true;
```

Inconvenientes del uso de hilos

- Condiciones de carrera
- Parámetros
- Excepciones asíncronas
- Rendimiento de los cambios de contexto

Tasks

Representa una operación asíncrona. Tienen un nivel de abstracción mayor y proporcionan más funcionalidad que los hilos, facilitando la programación paralela.

- TPL, PLINQ y otros elementos del lenguaje que facilitan paralelización de código se basan en ellas.

Su uso tiene dos beneficios principales:

- Uso más eficiente y escalable de recursos: los task se encolan automáticamente en el ThreadPool que maneja los hilos
- Mayor control de ejecución: soporta espera, cancelación, manejo robusto de excepciones…

Composición de tareas

Task y Task<TResult> tienen varios métodos para componer tareas, lo que permite implementar patrones típicos y mejorar el uso de las capacidades asíncronas del lenguaje.

- **Task.WhenAll:** Espera de forma asíncrona a que terminan varios Task o Task<Tresult>.
- **Task.WhenAny:** Espera de forma asíncrona a que uno o varios Task o Task<T> terminen.

- **TASK.DELAY:** Crea un objeto Task que termina tras un tiempo determinado.

Manejo de excepciones con tareas

Cuando se lanza una excepción desde una tarea, todas ellas se encapsulan en una ***AggregateException***, la cual se **propaga al hilo vinculado a la tarea**. El código que use alguna de los siguientes métodos puede tratar las excepciones mediante un try/catch: Wait, WaitAll, WaitAny, Result.

También se pueden tratar accediendo a la propiedad Exception de la tarea antes de ser eliminada.

Mecanismos de sincronización

THREAD.JOIN: Permite hacer que un hilo espere a la finalización de otro, evitamos condiciones de carrera.

La necesidad más típica de sincronización de hilos es por **acceso concurrente a recursos compartidos**. Evitar el acceso simultáneo a estos recursos se denomina **EXCLUSIÓN MUTUA.** Esto se aplica tanto a hilos como a tasks.

SECCIÓN CRÍTICA es un fragmento de código que accede a un recurso compartido que no debe ser accedido a la vez por más de un hilo.

Lock

Consigue que solo un hilo pueda ejecutar una sección crítica a la vez. Requiere especificar un objeto (referencia) como parámetro. Si un hilo ejecuta el lock sobre un objeto ya bloqueado, entonces se bloqueará hasta que se libere el objeto.

Debemos aplicar también lock a la mayor parte **de operaciones sobre un recurso compartido**, menos a las asignaciones de 32 bits, puesto que son atómicas. Otra alternativa es mediante la clase **Interlocked (System.Threading),** que es más eficiente que el lock.

LOCK(THIS): Puede originar esperas sin razón y afectar al rendimiento.

Mutex y semáforos

Llevan a cabo la sincronización entre procesos (o hilos).

- **MUTEX**: funciona similar a un lock pero con un coste de rendimiento mucho mayor
- **SEMÁFORO**: Permiten el acceso a n procesos concurrentes.

Interbloqueo

Todas las tareas están esperando por un evento que sólo otra puede causar, lo que lleva a que se bloqueen de forma permanente.

Condición de espera circular

Dadas ‘n’ tareas con el recurso R, que está esperando por el R0 que debe ser liberado mientras una tarea Tn, mientras Tn espera por Rn en la tarea Tn-1 mientras posee el recurso R0.

Evitando el interbloqueo

Para evitarlo, hay que impedir que se dé la condición de espera circular Necesitamos saber a priori los recursos que cada proceso necesita, pero esto no siempre es posible puesto que puede depender de su ejecución.

Thread Safety

Un programa es **THREAD-SAFE** si funciona correctamente cuando es utilizada por varios hilos a la vez.

Estructuras de datos thread-safe

La mayor parte de clases de propósito general no son thread-safe por cuestión de rendimiento.

Alternativas:

- Usar estructuras de datos especiales
- Realizar la sincronización con bloqueos de objetos thread-unsafe

Implementación de EEDD Thread-Safe

1. Partir de una estructura thread-unsafe
2. Crear una nueva clase con la misma interfaz
3. Realizar una composición a una instancia de la clase thread-unsafe
4. Realizar un bloqueo en el cuerpo de todos los métodos utilizando un objeto privado como monitor.

Implementación **sencilla** pero muy **ineficiente**. Las **lecturas concurrentes no requieren exclusión mutua**, las escrituras sí deben bloquear a otras lecturas y escrituras, esto lo podemos hacer mediante ReaderWriterLockSlim.

ReaderWriterLockSlim

Permite proteger un recurso compartido por varios hilos más eficientemente que lock.

Tres modos de bloqueo:

- **MODO DE LECTURA:** Cualquier número de hilos puede entrar al bloqueo.
- **MODO DE ESCRITURA:** Si un hilo entra en el bloqueo, no puede entrar ninguno más puede entrar en ningún modo.
- **MODO DE LECTURA ACTUALIZABLE:** **Bloqueo de lectura especial** que puede actualizarse a escritura sin tener que abandonar el acceso de lectura al recurso.

El código protegido por ReaderWriterLockSlim no puede tener llamadas recursivas por defecto (se puede cambiar pasando LockRecursionPolicy.SupportsRecursion al constructor), lo que disminuye la posibilidad de interbloqueos y mejora el rendimiento.

TPL y PLINQ - Simplificando el paralelismo

Task Parallel Library

- Simplifica la paralelización
- Escala y gestiona dinámicamente **los hilos en función del número de CPUs** y en función del Thread Pool
- Paralelización mediante **división de datos y tareas**
- **Declarativo**

Métodos más usados (System.Threading.Tasks.Parallel), ambos reciben un Action:

- **FOR**: crea potencialmente un hilo a partir de un índice de comienzo y final
- **FOREACH**: crea potencialmente un hilo por elemento

División de tareas con TPL

Clase Parallel, método **INVOKE**

- Recibe una lista de Actions
- Crea potencialmente un hilo por Action

Todos los métodos de TPL mencionados añaden sincronización para que en la siguiente instrucción ya hayan finalizado los hilos.

Paradigma Funcional

Los cambios de estado rompen la condición thread-safe de los programas. En el **paradigma funcional puro**, no hay modificaciones de estado gracias a la **transparencia referencial.**

.Net hace uso extensivo del paradigma funcional en el desarrollo de programas concurrentes por las ventajas en la creación de estos programas.

PLINQ

Implementación paralela de LINQ. Ofrece paralelismo de modo declarativo, elige dinámicamente el número de hilos y tiene una aproximación conservadora.

1. Parte los datos en segmentos
2. Ejecuta la consulta en paralelo
3. Cada hilo worker procesa un segmento distinto

Ley de Amdahl

Indica la mejora esperada máxima en un sistema cuando solo una parte del mismo es mejorada. Puede predecir la mejora máxima teórica en el uso de varios procesadores.

MapReduce

Modelo de programación de aplicaciones paralelas mediante división de datos usando los métodos Map y Reduce.

Tipado dinámica y metaprogramación

Tipado dinámico

Proceso de posponer la comprobación e inferencia de tipos en tiempo de ejecución.

VENTAJAS

- Mayor adaptabilidad y flexibilidad del código
- Mayor nivel de abstracción

DESVENTAJAS

- No se detectan errores en compilación
- Rendimiento pobre

En C#

dynamic pospone la comprobación e inferencia de tipos al tiempo de ejecución, por lo que el compilador no da error teniendo en cuenta que

- Cualquier tipo se puede convertir a dynamic y viceversa
- Cualquier operación se puede aplicar a dynamic

Duck Typing

Propiedad de la mayoría de los lenguajes con tipado dinámico por la cual el estado dinámico de un objeto determina qué operaciones pueden realizarse con él. Simplifica las jerarquías en el código del programa.

Multiple Dispatch

Duck typing es denominado a veces **LATE BINDING** (enlace tardío) puesto que posponen a tiempo de ejecución la resolución del método.

Una de las limitaciones del enlace dinámico es que supone usar **SINGLE DISPATCH**, es decir, sólo permite resolver un método dependiendo del tipo de un único objeto.

Cuando queremos que la resolución dependa de varios tipos necesitamos **MULTIPLE DISPATCH,** lo cual se lleva a cabo mediante multimétodos, no soportados en C#.

Reflexión

Capacidad de un sistema computacional de **razonar y actuar** sobre si mismo para adaptarse a condiciones cambiantes.

El dominio computacional se extiende con **la representación del propio sistema**, ofreciendo su estructura y semántica como datos computables en tiempo de ejecución.

INTROSPECCIÓN: Cuando se puede consultar la representación de un programa pero no modificarla. (ej. Obtener estructura de una clase)

INTERCESIÓN: Cuando se permite además modificar dicha representación (ej. Borrar o añadir un atributo a una clase)

```
Type type = figure.GetType();
PropertyInfo xProperty = type.GetProperty("X");
xProperty.SetValue(figure, (int) xProperty.GetValue(figure,null) + 10, null);
```

Tipos de reflexión:

Estructural

- La información es la estructura de un programa.
- C# ofrece introspección estructural pero solo intercesión estructural en objetos de la clase ExpandoObject.
    - Se permite añadir atributos y métodos a este tipo de objeto
    - Los métodos se presentan como delegados.
    - ExpandoObject deriva de IDictionary<string,object>

- Se puede aplicar duck typing sobre estos objetos gracias a dynamic.

```
dynamic person = new ExpandoObject();
person.FirstName = "John";
foreach(var item in dict)
        ((IDictionary<string,object>)person)[item.Key] = item.Value;
```

Suponiendo que dict es un diccionario que contiene los nombres de los atributos y los valores.

De comportamiento

C# ofrece un mecanismo limitado de reflexión de comportamiento, el cual permite crear objetos derivados de DynamicMetaObject y poder modificar su comportamiento relativo a: invocación de métodos, obtención/modificación de un miembro…

Anotaciones o atributos

Los **atributos** (.Net) o **anotaciones** (Java) son metadatos que dan información adicional de elementos del lenguaje y añaden información declarativa que puede ser utilizada para obtener un comportamiento distinto.

Se basa en el principio de "**Separation of Concerns**", que consiste en dividir el código en módulos, evitando el entremezclado de código de distintas funcionalidades

Están escritos en el propio lenguaje (Serializable, Obsolete, Test…)

Atributos en C#

Se pueden aplicar a Ensamblados, módulos, clases, interfaces… (básicamente a casi todo)

[Test], [Obsolete("msg") …

Custom attributes

Se crean en el mismo lenguaje de programación y ofrecen una **pseudoextensión** del lenguaje

1. Definimos el uso del atributo (clase, método…)
2. Derivamos de la clase Attribute
3. Implementamos la clase (el atributo)
4. Aplicamos el atributo a un programa
5. Se crea otra aplicación que procesa los atributos

Generación Dinámica de Código

Capacidad de **generar programas** (o parte) en **tiempo de ejecución** (programas que generan programas)

Se puede usar para generar dinámicamente el código de nuevos métodos o clases (**intercesión estructural**) a partir de:

- El propio programa: **Adaptiveness**
- El usuario (directa o indirectamente): **Adaptabilllity**

También llamada **programación generativa**.

En C#

Proporciona la característica de compilador como servicio a través de Roslyn, permitiendo usar el compilador desde nuestro propio código. También permiten la generación dinámica de código y su ejecución a partir de Strings, ficheros y árboles de sintaxis abstracta. El código se puede compilar, cargar e invocar en tiempo de ejecución.

Metaprogramación

Capacidad de escribir programas que escriban o manipulen otros programas. Conceptos relacionados con la meta-programación: Reflexión, generación dinámica de código, anotaciones o atributos, tipado dinámico.