TPP-Resumen-21-22.pdf

AlbertoM02

Tecnologias y Paradigmas de la Programacion

2º Grado en Ingeniería Informática del Software

Escuela de Ingeniería Informática
Universidad de Oviedo

Que no te escriban poemas de amor
cuando terminen la carrera ►►►►►►

(a nosotros por suerte nos pasa)

WUOLAH

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de pagar

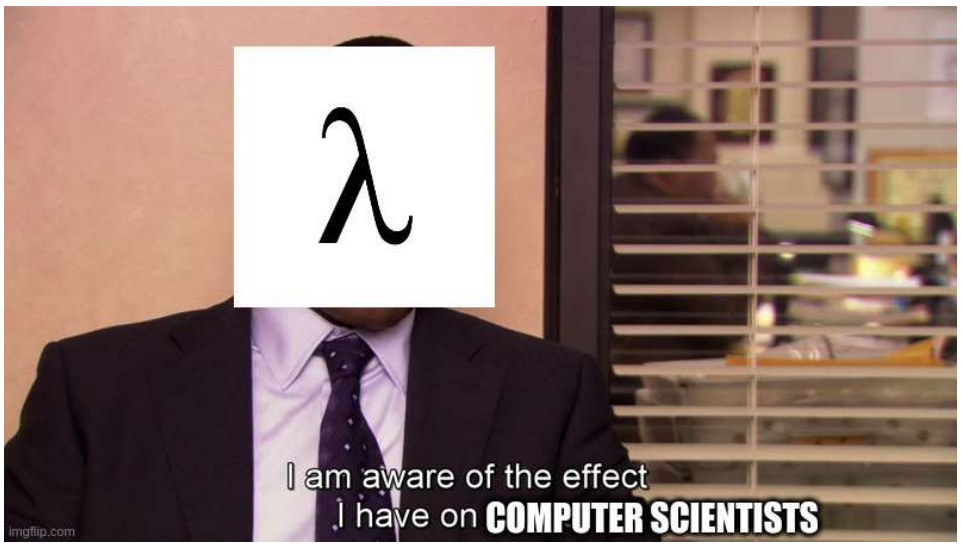
Llegó mi momento de despedirte
Tras años en los que has estado mi lado.

Siempres me has ayudado
Cuando por exámenes me he agobiado

Oh Wuolah wuolitah
Tu que eres tan bonita

Tecnologías y Paradigmas de la Programación

TPP para los amigos



Lenguajes y Paradigmas de la Programación

Tema 1

Lenguaje de Programación

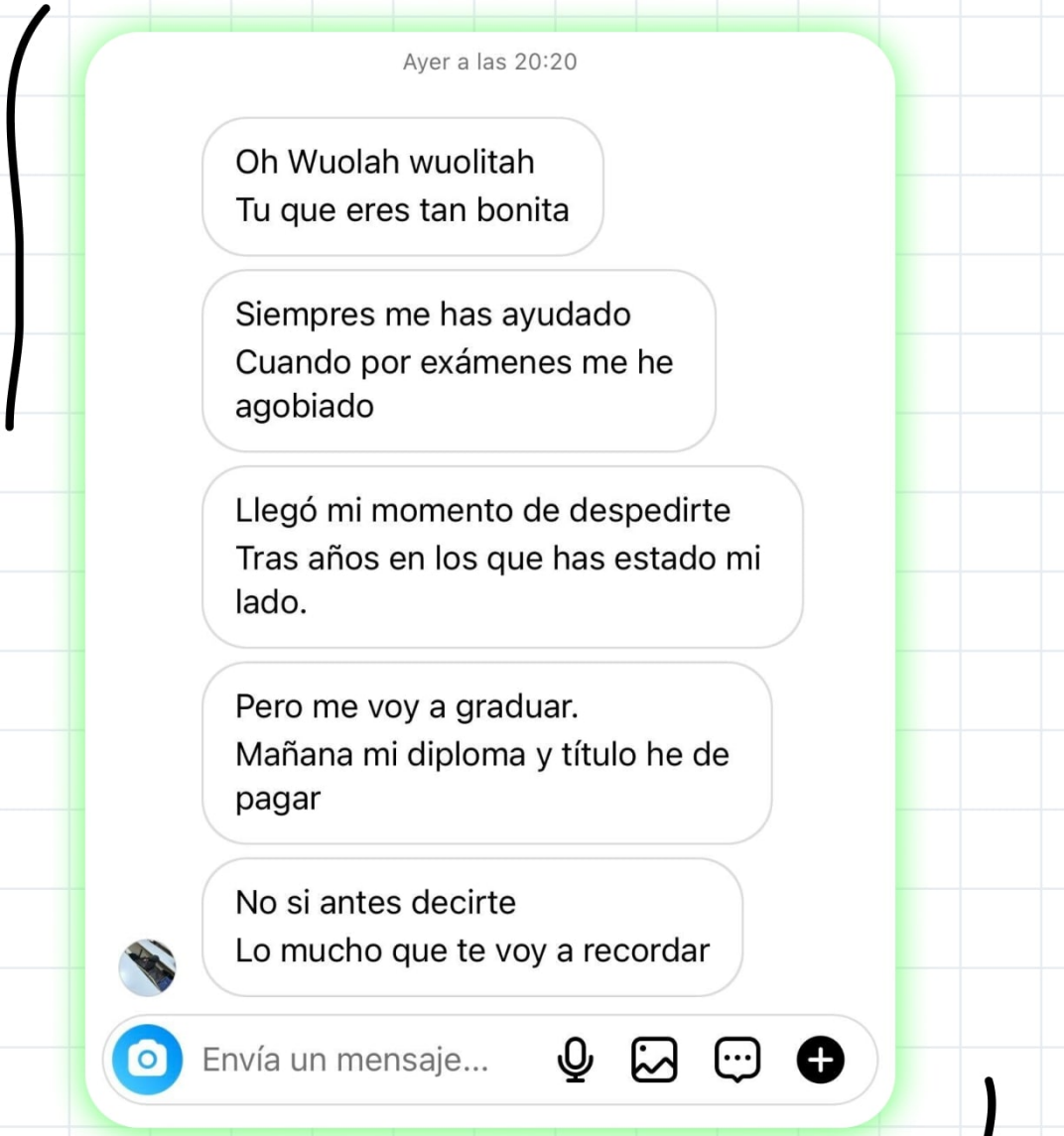
- Un lenguaje de programación es un lenguaje artificial utilizado para escribir instrucciones, algoritmos o funciones que pueden ser ejecutadas por un ordenador
- Es un lenguaje que permite acercar el nivel de abstracción humano al nivel de abstracción de una máquina

Traductor, Compilador e Intérprete

- Un traductor es un programa que procesa un texto fuente (origen) y genera un texto objeto (destino)
- Un compilador es un traductor que transforma código fuente de un lenguaje de alto nivel al código fuente de otro lenguaje de bajo nivel (Un compilador es, por tanto, un caso particular de un traductor)
- Un intérprete ejecuta las instrucciones de los programas escritos en un lenguaje de programación

Que no te escriban poemas de amor cuando terminen la carrera 🚩🚩🚩🚩🚩🚩

(a nosotros por suerte nos pasa)



WUOLAH

Clasificación de lenguajes

1. Nivel de abstracción
2. En función de su dominio
3. Soporte de concurrencia
4. En función de su implementación
5. En función de cuando se realizan las comprobaciones de tipo
6. En función de como se representa su código fuente

Nivel de abstracción

- Suele medir en qué medida el lenguaje está más nivel de abstracción del próximo al lenguaje o sistema cognitivo humano (alto nivel) o más próximo al ordenador (bajo nivel)
- El lenguaje ensamblador y código máquina están considerados como lenguajes de bajo nivel
- El C es considerado como un lenguaje de medio nivel por algunos autores
- El resto de lenguajes de programación son comúnmente considerados como lenguajes de alto nivel
- Los lenguajes específicos del dominio de (DSLs) suelen tener aún un mayor nivel de abstracción

En función del dominio.

- Los DSL son lenguajes específicos de un dominio.
- Los lenguajes de propósito General pueden ser utilizados para resolver cualquier problema computacional.
- Los lenguajes específicos han aumentado su uso en los últimos años debido al modelado específico del dominio y desarrollo dirigido por modelos.

Soporte de concurrencia.

Aquellos lenguajes que permiten la creación de programas mediante un conjunto de procesos o hilos que interaccionan entre sí. Y que pueden ser ejecutados de forma paralela.

- En un Único procesador. Intercalando la ejecución de cada 1 de los procesos.
- En paralelo, asignando cada proceso o hilo a procesadores distintos que pueden estar en una misma máquina o distribuidos en una red de computadores.

En función de su implementación.

Compilados o interpretados. Esa clasificación no es necesariamente excluyente.

- Un compilador JIT transforma el código de interpretar en código nativo propio de la plataforma donde se ejecuta el intérprete. Justo antes de la ejecución.
- Si esa transformación se realiza previamente a la ejecución, se denomina AoT

En función de cuándo se realizan las comprobaciones de tipo

Las comprobaciones de tipo Pueden ser estáticas o dinámicas. Esta no es una clasificación excluyente.

- La comprobación estática de tipos suele ofrecer detección temprana de errores y un mayor rendimiento.
- La comprobación dinámica suele ir ligada a una mayor adaptabilidad dinámica. y a una meta programación dinámica.

En función de cómo se representa su código fuente.

Puede tratarse de lenguajes visuales o textuales

- Los lenguajes visuales representan las entidades de su dominio mediante una anotación visual en lugar de textual
- Existen numerosos lenguajes visuales específicos de un dominio
- Los lenguajes textuales pues son ficheros de texto para contener su código fuente

Paradigmas de programación

Un paradigma de programación es una aproximación para construir programas caracterizada por la abstracción y conceptos utilizados para representar dichos programas

- El paradigma de programación se usa también para clasificar lenguajes de programación
- Un lenguaje de programación puede seguir más de un paradigma
- Un paradigma de programación no constituye una característica de un lenguaje

Imperativo vs declarativo

Determinados autores los consideran sí paradigmas

Más bien suponer una clasificación de los paradigmas en algunos casos incluso de lenguajes

- Un paradigma imperativo describe los programas en términos de sentencias que cambian el estado del programa
- Los lenguajes declarativos se basan en escribir programas que especifican que se quiere obtener intentando no especificar como

Principales paradigmas

1. Estructurado basado en procedimientos: paradigma imperativo en el que se utilizan 3 estructuras de control secuencial condicional e iterativa, pudiendo agrupar estas en procedimientos o subrutinas
2. Orientado a objetos: paradigma comúnmente imperativo que utiliza los objetos, Unión de datos y métodos, como principal abstracción, definiendo programas como interacciones entre objetos
3. Funcional: paradigma declarativo basado en la utilización de funciones que manejan datos inmutables
4. Lógico: paradigma declarativo basado en la programación de ordenadores mediante lógica matemática

Estructurado basado en procedimientos

Este paradigma es comúnmente llamado simplemente imperativo

Viene de la unión de 2 paradigmas históricos: el paradigma estructurado y el paradigma procedural

- Define el procedimiento a su rutina como el primer mecanismo de descomposición
- Incorporó la programación de estructurada
- Incluye el concepto de ámbito como previniendo el acceso indebido a variables
- Un procedimiento que devuelve un valor se define como una función

Paradigma estructurado: paradigma en el que se utilizan 3 estructuras de control; Secuencial, condicional e iterativa

- Se evitan las instrucciones de salto, tanto condicional como incondicionales
- Se aumenta la legibilidad y mantenibilidad de los programas
- Las estructuras de control se pueden anidar

Orientación a objetos

- Utiliza los objetos, unión de datos y métodos, como principal abstracción, definiendo programas como interacciones entre objetos.
- Se basa en la idea de modelar objetos reales, o introducidos en diseño, mediante la condición de objetos software
- Un programa está constituido por un conjunto de objetos pasándose mensajes entre sí típicamente es un paradigma imperativo
- Los elementos propios de este paradigma son el encapsulamiento, herencia, polimorfismo y enlace dinámico
- Modelos computacionales:
    - Basados en clases:
        - Todo objeto tiene que ser instancia de una clase
        - Clases la clase es el tipo de objeto
        - Las clases definen la estructura y comportamiento de sus instancias
    - Basados en prototipos:
        - No existe el concepto de clase, los objetos son la única entidad
        - Se define un objeto prototipo y el resto de objetos con una estructura similar se clonan a partir de este

Paradigma funcional

Paradigma declarativo basado en la utilización de funciones que manejan datos inmutables

- Un programa se define mediante un conjunto de funciones invocándose entre sí
- Las funciones no generan efectos colaterales
    - El valor de una expresión depende exclusivamente de los valores de los parámetros
    - Devolviendo siempre el mismo valor en función de estos
- Se hace uso exhaustivo de la recursividad, en lugar de la iteración
- Los lenguajes funcionales puros no utilizan ni la asignación ni la secuenciación de instrucciones
- Sus orígenes se remontan al cálculo lambda

Paradigma lógico

Paradigma declarativo basado en la programación de ordenadores mediante lógica matemática

- El programador describe un conocimiento mediante reglas lógicas y axiomas
- Un demostrador de teoremas con razonamiento hacia atrás resuelve problemas a partir de consultas

Elementos del paradigma orientado a objetos en C#

Tema 2

Paradigma orientado a objetos

- Utiliza los objetos, unión de datos y métodos, como principal abstracción definiendo programas como interacciones entre objetos
- Se basa en la idea de modelar objetos reales, o introducidos en diseño mediante la codificación de objetos software
- Un programa está constituido por un conjunto de objetos pasándose mensajes entre sí

Abstracción

- Abstracción: expresa las características esenciales de un objeto, las cuales distinguen al objeto de los demás

Encapsulamiento

- Encapsulamiento: proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento
- La ocultación de información permite discernir entre qué partes de la abstracción están disponibles al resto de la aplicación y qué partes son internas a la abstracción
- Para ello, los lenguajes de programación ofrecen diversos niveles de ocultación para sus miembros
- Cada objeto está aislado del exterior y expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase

Propiedades

- C# ofrece el concepto de propiedad para acceder al estado de los objetos como si del atributo se tratase, obteniendo los beneficios del encapsulamiento
- Las propiedades pueden ser de lectura y / o escritura
- Las propiedades pueden catalogarse con todos los niveles de ocultación, ser de clases como ser abstractas o sobrescribirse

Modularidad

- Propiedades que permiten subdividir una aplicación en partes más pequeñas, siendo cada una de ellas tan independiente como sea posible
- Cada módulo ha de poder ser compilado por separado para ser utilizados en diversos programas
- Distintos elementos pueden constituir un módulo: funciones y métodos, clases y tipos, espacio de nombres y packages, componentes

Acoplamiento y cohesión

- Acoplamiento: nivel de interdependencia entre módulos
- Cohesión: nivel de uniformidad y relación que existe entre las distintas responsabilidades un módulo

En desarrollo software el bajo acoplamiento y elevada cohesión favorecen a la reutilización y mantenibilidad del software

Sobrecarga de métodos

- La sobrecarga de métodos permite dar distintas implementaciones a un mismo identificador del método
- En C#, para sobrecargar un método es necesario que cada método sobrecargado difiera de los otros en al menos 1 de los siguientes puntos:
  - El número de parámetros
  - El tipo de algunos de sus parámetros
  - El paso de algunos de sus parámetros (valor, ref o out)
- En Java el último punto no es aplicable

Sobrecarga de operadores

- La sobrecarga de operadores permite modificar la semántica de los operadores del lenguaje
- C# ofrece sobrecarga de operadores incluyendo ++, --, [ ]…
- Aunque C# ofrece sobrecarga de operadores está apenas se usa

Herencia

- La herencia es un mecanismo de reutilización de código
- El estado de una instancia derivada está definido por la unión de las estructuras de las clases base y derivadas
- El conjunto de mensajes que puede aceptar un objeto derivado es la unión de los mensajes de su clase base y derivada

Polimorfismo

- Es un mecanismo de generalización, que hace que la abstracción más general pueda representar abstracciones más específicas
- Para ello la conversión ascendente en la jerarquía es automática
- Enlace dinámico:
  - Los métodos heredados se pueden especializar en las clases derivadas
  - Si queremos que se llame al método real implementado por el objeto debemos hacer uso del enlace dinámico
  - Para introducir este enlace dinámico en C# debemos poner previamente la palabra reservada virtual en el método que reciba el mensaje
  - También debemos redefinir su funcionalidad utilizando la palabra reservada override en los métodos derivados

Herencia múltiple

- La herencia múltiple produce dos conflictos
    - Coincidencia de nombres: se produce cuando se hereda de 2 o más clases un miembro con igual identificador se produce una ambigüedad en su acceso
    - Herencia repetida: se produce cuando se hereda más de una vez de una clase por distintos caminos

Interfaces

- En ocasiones necesitamos que un tipo sea un subtipo de 2 o más súper tipos
- Una interfaz es un conjunto de mensajes públicos, que ofrecen un conjunto de clases
- En C# este concepto se ofrece como un tipo

Objetivos del manejo de excepciones

1. Reutilizable: en los diversos contextos en los que se emplee la abstracción, el manejo de errores puede ser totalmente distinto
2. Robusto: el sistema deberá obligar al programador a gestionar el posible error de forma distinta al flujo general de ejecución
3. Extensible: en función del uso de una abstracción un error:
    a. Puede transformarse en otros errores
    b. Puede manejarse corrigiendo el posible error

Excepciones: una excepción es un evento que se produce en un momento de ejecución y que impide que la ejecución prosiga con su flujo normal, el mecanismo de manejo de excepción se basa en la separación de la abstracción que detecta el error y las distintas abstracciones que manejan la excepción

Asertos: los asertos son condiciones que se han de cumplir en la correcta ejecución de un programa, en caso de producirse se detiene la ejecución del programa. Los asertos no se deben utilizar para detectar errores en tiempo de ejecución ya que no son ni reutilizables ni extensibles.
Los asertos se pueden utilizar para detectar aquellas situaciones que nunca deben ocurrir y en caso de que se ocurran se trataría de un error de implementación que debemos correr estos asertos deberían deshabilitarse una vez la aplicación haya sido probada exhaustivamente

Genericidad

- La genericidad es la propiedad que permite construir abstracciones modelo para otras abstracciones
- Ofrece dos beneficios principales:
    - Una mayor robustez
    - Un mayor rendimiento
- La genericidad acotada permite hacer que los tipos genéricos sean más específicos

Fundamentos del paradigma funcional

Tema 3

Paradigma funcional

- Paradigma declarativo basado en la utilización de funciones que manejan datos inmutables
    - Los datos nunca se modifican
    - En lugar de cambiar un dato se llama a una función que devuelve el dato modificado sin modificar el original
- Un programa se define mediante un conjunto de funciones invocándose entre sí
- Las funciones no generan efectos secundarios
    - El valor de una expresión depende exclusivamente de los valores de los parámetros
    - Devolviendo siempre el mismo valor en función de esto

Cálculo lambda

- El cálculo lambda es un sistema formal basado en la definición de funciones y su aplicación
- El cálculo lambda se considera como el lenguaje más pequeño universal de computación
    - Es universal porque cualquier función computable puede ser expresada y evaluada usando este formalismo

Expresiones lambda

- En el cálculo lambda, una expresión lambda se define como
    - una abstracción lambda donde x es una variable -parámetro- y M es una expresión lambda -cuerpo de la función-
    - una aplicación MN dónde M y N son expresiones lambda
- Ejemplos de abstracciones
    - La función identidad f(x)=x puede representarse como la expresión lambda λx.x
    - La función identidad f(x)=x+x puede representarse como la expresión lambda λx.x+x (x+x no es una expresión lambda pero buscamos simplicidad en el ejemplo)

Aplicación (reducción-β)

- La aplicación de una función representa su invocación
- La aplicación se define del siguiente modo
  (λx.M)N →M[x:=N] (o M[N/x])
  Siendo x una variable y M y N expresiones lambda y M[x:=N] (o M[N/x]) representa M donde todas las apariciones de x son sustituidas por N
- Esta sustitución se denomina reducción-β
- Ejemplos de aplicación:
    - (λx.x+x)3 →3+3
    - (λx.x) λy.y*2 → λy.y*2

Teorema de Church-Rosser

- En algunos términos lambda se puede aplicar múltiples reducciones beta
  - (λx.x) (λy.y*2) 3
- El teorema de Church-Rosser establece que el orden en el que se hagan estas reducciones no afecta al resultado final
- Por tanto los paréntesis se usan normalmente para delimitar términos lambda no indican preferencias

Variables libres y ligadas

- En abstracción λx.xy se dice que
  - La variable x está ligada
  - La variable y es libre
- En una sustitución solo se sustituyen las variables libres
  (λx.x(λx.2+x)y)M→x(λx.2+x)y[x:=M]=M(λx.2+x)y
  - La segunda x no se sustituye ya que representa una variable distinta
- Para evitar estos conflictos de nombre se creó la conversión-α
  - Todas las apariciones de una variable ligada en una misma abstracción se pueden renombrar a una nueva variable
    (λx.x(λx.2+x)y)M λ (λx.x(λz.2+z)y)M → x(λz.2+z)y[x:=M] = M(λz.2+z)y
    λM(λx.2+x)y

Curry-Howard

- El isomorfismo o correspondencia de Curry-Howard establece una relación directa entre problemas software y demostraciones matemáticas
  - Estableciendo una correspondencia entre la lógica y la computación
  - En cálculo lambda, esta correspondencia es directa
- Esto implica que:
  - Existe una correspondencia entre tipos y proposiciones
  - Existe una correspondencia entre programas y evidencias que demuestran la proposición descrita por su tipo

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 basa en lenguajes funcionales
- Añaden un lenguaje para realizar demostraciones mediante deducción natural
- Permiten la extracción de programas: generan el código en distintos lenguajes una vez realizada la demostración

Funciones entidades primer orden

- El paradigma funcional identifica las funciones como entidades de primer orden, igual que el resto de valores: funciones de primer orden
- Esto significa que las funciones son un tipo más, pudiéndose instanciar variables de tipo función para, por ejemplo:
    - Asignar las estructuras de datos
    - Pasarlas como parámetros a otras funciones
    - Retornarlas como valores de otras funciones
- Una función se dice que es una función de orden superior si:
    - Recibe alguna función como parámetro
    - O retorna una función como resultado

Funciones de orden superior

- La función doble aplicación, que vimos anteriormente es una función de orden superior
  λf.(λx.f(fx))
- Recibe una función como parámetro y, dada otra expresión, la utiliza para pasársela a la función y, su resultado, se lo devuelve a pasar a la función

Delegados

- En C # las funciones son entidades de primer orden gracias a los delegados
- Un delegado constituye un tipo que representa un método de instancia o de clase
- Las variables de tipo delegado representan un modo de referencia un método
- Por tanto:
    - 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

Tipos de delegados predefinidos

- Los más utilizados son Func, Action y Predicate
    - Func<T>: método sin parámetros que retorna un T
    - Func<T1, T2>: Método con un parámetro T1 que retorna un T2
    - …
    - Action: método sin parámetros ni retorno
    - Action<T>: Método con un parámetro sin retorno
    - …
    - Predicate<T>: método que retorna un bool y recibe un T

Delegados anónimos

- El tener que implementar un método para pasarlo como parámetro posteriormente es tedioso
- En la programación funcional es como un poder escribir la función en el momento de pasársela

- La primera aproximación que ofreció C # para ello fueron los delegados anónimos
    - Sintaxis para definir una variable delegado indicando sus parámetros y su cuerpo
    - Sigue siendo una sintaxis un tanto prolija

Expresiones lambda

- Las expresiones lambda permiten describir el cuerpo de funciones completas como expresiones siguiendo la aproximación del cálculo lambda, siendo a su vez una mejora de los delegados anónimos
- Sintaxis
    - Se especifican los parámetros separados por comas si los hay
        - Opcionalmente se pueden anteponer los tipos
        - Si hay más de un parámetro se debe utilizar paréntesis
        - Sí hay cero parámetros se indica con ()
    - El símbolo => indica la separación de los parámetros y del cuerpo de la función
    - Si el cuerpo tiene varias sentencias se separa con;
    - En el cuerpo se utiliza return para devolver valores
    - Si el cuerpo es una única sentencia no es necesario escribir return ni llaves

Clausulas

- Una cláusula es una función de primer orden junto con su ámbito: una tabla que guarda las referencias a sus variables libres
- Las variables libres de una cláusula representan estado
    - Este estado puede, además, estar oculto cuando el ámbito de la variable finaliza
    - Por tanto, pueden representar objetos
- Las cláusulas también pueden representar estructuras de control

Currificación

- El cálculo lambda es un lenguaje de programación universal: cualquier función computable puede representarse en el
    - Números y operaciones enteras y booleanas
- No obstante, la definición de abstracción solo define un único parámetro para las funciones lambda
- La currificación es la técnica para transformar una función de varios parámetros en una función que recibe un único parámetro
    - La función recibe un parámetro y retorna otra función que se puede llamar con el segundo parámetro
    - Esto puede repetirse para todos los parámetros de la función original
    - La invocación se convierte en llamadas a funciones encadenadas
- Cuando las funciones están currificadas es posible realizar su aplicación parcial
- La aplicación parcial consiste en pasar un número menor de parámetros en la invocación de la función (el resultado es otra función con un número menor en aridad)

- Esto le da gran potencial al lenguaje especialmente cuando los operadores son funciones

Continuaciones

- Una continuación representa el estado de computación en un momento de ejecución
- El estado de computación normalmente está compuesto por, al menos:
    - El estado de la pila de ejecución
    - La siguiente instrucción de ejecutar
- Los lenguajes que ofrecen continuaciones son capaces de almacenar su estado de ejecución y recuperarlo posteriormente

Generadores

- Un generador es una función que simula la devolución de una colección de elementos sin construir toda la colección devolviendo un elemento cada vez que la función es invocada
- Una implementación de generadores es mediante continuaciones
- El hecho de no construir toda la colección hace que sea más eficiente
- Un generador es una función que se comporta como un iterador

Evaluación perezosa

- La evaluación perezosa es la técnica por la que se demora la evaluación de una expresión hasta que esta es utilizada
- Relativo al paso de parámetros en la mayoría de lenguajes ofrecen paso ansioso
- Los beneficios de la evaluación perezosa son:
    - Un menor consumo de memoria
    - Un mayor rendimiento
    - La posibilidad de poder crear estructuras de datos potencialmente infinitas
- La implementación de un procesador de lenguaje que soporte evaluación perezosa es más compleja

Transparencia referencial

- Una expresión se dice que es referencialmente transparente si esta se puede sustituir por su valor sin que se cambie la semántica del programa
- Mientras que en matemáticas todas las funciones son referencialmente transparentes en programación no
- La transparencia referencial es uno de los pilares del paradigma funcional
- Cuando es ofrecida en un lenguaje, se dice que es funcional puro
- Elementos de un lenguaje que hacen que no ofrezca transparencia referencial son:
    - Variables globales mutables
    - Asignaciones destructivas
    - Funciones impuras

Variables globales y asignaciones

- Las variables mutables fuera del ámbito de una función hacen que no se obtenga transparencia referencial
- Con las asignaciones sucede lo mismo la evaluación de una variable depende de sus asignaciones previas

Funciones puras

- La utilización de funciones que no son puras, implican opacidad referencial
- Una función es pura cuando:
    - Siempre devuelve el mismo valor ante los mismos valores de los argumentos (no depende de un estado dinámico, ni un dispositivo de entrada salida)
    - La evaluación de una función no genera efectos secundarios (no actualiza variables globales, ni dispositivos entrada y salida, ni estados dinámicos)

Beneficios

- Los beneficios de la transparencia referencial son:
    - Se puede aplicar el razonamiento matemático a los programas
    - Se pueden realizar transformaciones en los programas para simplificarlos optimizarlos o paralelizarlos

Fundamentos de la programación concurrente y paralela

Tema 4

Arquitecturas multinúcleo

- La tendencia actual es que los ordenadores amplíen el número de núcleos de procesamiento en lugar de la frecuencia de reloj
- Los microprocesadores multinúcleo incluyen más de un procesador en una misma CPU
- Ofrecen computación paralela no solo a nivel de proceso, sino también a nivel de hilo
- Las arquitecturas multinúcleo poseen una memoria compartida

Memoria distribuida

- En los sistemas multiprocesadores con memoria distribuida, cada procesador posee una memoria privada
- Si un procesador requiere datos de otros, tiene que comunicarse con éste a través de un canal

Programación concurrente

- Debido a esta tendencia, cada vez es más importante realizar programación concurrente y paralela
- Concurrencia es la propiedad por la que varias tareas se puede ejecutar simultáneamente y potencialmente interactuar entre sí

Programación paralela

- Paralelismo es un caso particular de la concurrencia, en el que las tareas se ejecutan de forma paralela
- El paralelismo comúnmente enfatiza la división de un problema en partes más pequeñas
- La programación concurrente comúnmente enfatiza la interacción entre tareas

Proceso

- Un proceso es un programa en ejecución
- Consta de instrucciones, estado de ejecución y valores de los datos en ejecución
- En los sistemas de memoria distribuida, las tareas concurrentes en distintos procesadores son procesos

Hilo

- Un proceso puede constar de varios hilos de ejecución
- Un hilo de ejecución es una tarea de un proceso que puede ejecutarse concurrentemente, compartiendo la memoria del proceso, con el resto de sus hilos
- Propio de cada hilo es el contador de programa, la pila de ejecución y el valor de los registros

- En los procesadores multinúcleo, los hilos pueden ser tareas paralelas de un mismo proceso

Paralelización de algoritmos

- Existen dos escenarios típicos de paralelización
    - Paralelización de tareas: tareas independientes pueden ser ejecutadas concurrentemente
    - Paralización de datos: ejecutar una misma tarea que computa porciones de los mismos datos

Pipeline

- Existe un modelo híbrido de los dos anteriores denominado pipeline
- Si queremos calcular hashes de cadenas encriptadas, podemos ejecutar dos tareas paralelas: encriptar la cadena y calcular hash

Hilos (Threads)

Context Switch

- El contexto de una tarea es la información que puede ser guardada cuando éste es interrumpido para que luego pueda reanudarse su ejecución
- El cambio del contexto es la acción de almacenar/restaurar el contexto de una tarea para que pueda ser reanudada su ejecución
- Esto permite la ejecución concurrente de varias tareas en el mismo proceso
- El cambio de contexto requiere:
    - Tiempo de computación para almacenar y restaurar el contexto de varias tareas
    - Memoria adicional para almacenar los distintos contextos
- Por tanto, la utilización de un número elevado de tareas, en relación con el número de procesadores, puede conllevar una caída global del rendimiento

Thread Pooling

- No solo el cambio de contexto implica un coste computacional en aplicaciones concurrentes
- La creación y destrucción de hilos es un proceso que también implica un coste computacional y de memoria
- Por todo ello, se debe:
    - Limitar el número máximo de hilos creados por un proceso, En relación con el número de procesadores y otros recursos de la memoria
    - Minimizar el número de hilos creados
- Esta técnica también se utiliza con conexiones a bases de datos

Foreground & Background Threads

- Los hilos que hemos creado ahora con la clase Thread han sido hilos foreground
    - La aplicación no finalizará hasta que acabe la ejecución de todos estos hilos

- Un hilo background es aquel que será terminado cuando no queden hilos foreground en ejecución
    - Normalmente son proveedores de servicios
    - No confundir con hilos secundarios o Workers

Inconvenientes del uso de hilos

- Condiciones de carrera: debemos esperar explícitamente hasta que todos los hilos hayan terminado de realizar sus cálculos
- Parámetros: sin parámetros o solo un objeto, Variables libres compartidas
- Excepciones asíncronas: las excepciones originalmente en un hilo no son capturadas por bloques try catch pertenecientes a un hilo diferente
- Rendimiento de los cambios de contexto: no hay optimización automática del número de hilos creados

Tasks

Varios de los problemas que hemos visto con hilos se resuelven mediante una abstracción la tarea. Tiene un nivel de abstracción mayor y proporciona más funcionalidades que los hilos, facilitando la programación paralela

- Una Task representa una operación asíncrona y su uso tiene dos beneficios principales
    - Uso más eficiente y escalable de recursos: las Task se encolan automáticamente en el ThreadPool, que optimiza transparentemente el número real de hilos usados y hace balanceo de carga para maximizar el uso de recursos del sistema
    - Mayor control de ejecución: el API de Task soporta espera, cancelación, continuación, manejo robusto de excepciones, consulta detallada del estado planificación y más características

Creación y ejecución explícita de tareas

- Una tarea que no devuelve valores está representada por la clase System.Threading.Tasks.Task
- Una tarea que devuelve valores está representada por la clase System.Threading.Tasks.Task<TResult>
- Una instancia detrás maneja todos los detalles de la infraestructura, proporcionando métodos y propiedades al hilo que la crea y ejecuta durante toda su vida útil
- Para crear una task es necesario suministrar un delegado que encapsule el código a ejecutar
- Finalmente, wait para la ejecución del hilo que lo llamaba hasta que su Task termine

Composición de tareas

- Task y Task<TResult> tienen varios métodos para componer tareas
- Esto permite incrementar patrones típicos y mejorar el uso de las capacidades asíncronas del lenguaje

Manejo de excepciones con tareas

- Cuando una tarea lanza una o más excepciones, todas ellas se encapsulan en una excepción de tipo AggregateException
- Esta excepción se prolonga el hilo vinculado a la tarea, que normalmente es el que espera a que termine o bien el que accede a su propiedad result
- El código que llama a 1 de los siguientes métodos de la tarea obtendrá las excepciones que se produzcan en la ejecución de la misma y podrá tratarlas normalmente en un bloque try catch
- También se pueden tratar excepciones de una tarea accediendo a la propiedad exception de la misma antes de que sea eliminada por el recolector de basura

Paso asíncrono de mensajes

El primer método para crear hilos es el paso de mensajes asíncronos con cada mensaje crea un hilo.

- En las últimas versiones de C# se ha introducido las palabras clave async y await
- Permiten escribir código asíncrono de manera fácil e intuitiva
- Ambas palabras reservadas se utiliza en combinación con la otra
- El antiguo modelo de paso del mensaje es asíncrono utilizaba llamadas menos intuitivas
- Un método async normalmente devuelve una Task o una Task<TResult> que representa el trabajo que se está realizando en el método
- Esta tarea tiene información que puede usar quién ha invocado el método asíncrono
- El operador await se utiliza sobre la task devuelta por un método async
- Await cede el control al invocador del método y solo puede utilizarse en métodos asíncronos

Herramientas de sincronización

Sincronización de hilos

- En ocasiones, varios hilos tienen que colaborar entre sí para conseguir un objetivo común
- Puesto que el orden de ejecución es no determinista, es necesario utilizar mecanismos de sincronización de hilos para evitar condiciones de carrera
- Un primer mecanismo básico de sincronización de hilos que ya hemos utilizado es Thread.Join
- No obstante, la necesidad más típica de sincronización de hilos es por acceso concurrente a recursos compartidos
- Evitar el uso simultáneo de recursos comunes se denomina exclusión mutua
- Esto se aplica a aplicaciones con varios hilos, tanto si se implementan con creación explícitas de hilos como si se utilizan Task

Recurso compartido

- Cuando varios hilos pretenden acceder al mismo recurso puede ocurrir que lo llamen cuando esa información ha cambiado

- Una sección crítica es un fragmento de código que accede a un recurso compartido que no debe ser accedido concurrentemente por más de un hilo de ejecución
- La sincronización de hilos debe usarse para conseguir la exclusión mutua

Lock

- La principal técnica utilizada para sincronizar y los C# es la palabra reservada lock
- Consigue que únicamente un hilo pueda ejecutar una sección de código Simultáneamente => exclusión mutua
- lock requiere especificar un objeto o referencia como parámetro
- El objeto modela un padlock:
    - Un hilo bloquea el objeto
    - Ejecuta la sección crítica
    - Sale del bloqueo
- Si otro hilo ejecuta el lock sobre un objeto que está bloqueado, entonces se pondrá en modo de espera y se bloqueará hasta que el objeto sea liberado

Asignaciones

- Hemos visto cómo la escritura de recursos compartidos requiere sincronización esto también sucede en las asignaciones de variables y atributo
- No todas las asignaciones son atómicas en .net
- Por tanto, las asignaciones multihilo de una misma variable deben sincronizarse
- Una alternativa es utilizar lock
- Otra alternativa es utilizar los métodos de la clase Interlocked

Mutex y Semaphore

- Hasta ahora hemos visto sincronización entre hilos
- La sincronización entre procesos se puede conseguir con mutex y semáforos
- Un Mutex es un mecanismo de sincronización de procesos
    - Su funcionamiento es similar a lock
    - Su coste de rendimiento es de, aproximadamente, unas 50 veces mayor a lock
- Los semáforos, adicionalmente, permiten el acceso a n procesos concurrentes
    - Se suelen utilizar para limitar la concurrencia

Interbloqueo

- Se produce un interbloqueo entre un conjunto de tareas si todas y cada una de ellas están esperando por un evento que solo puede causar otra
- Todas las tareas se bloquean permanentemente
- El caso más común es el acceso a recursos compartidos

Condición de espera circular

- Esta condición se produce cuando dadas n tareas T0…Tn , T0 Posee el recurso R1 pero espera por el R0 que debe ser liberado por la tarea Tn… mientras Tn espera por el recurso Rn poseído por la tarea Tn-1 mientras posee el recurso R0
- Esta condición causa el interbloqueo de las tareas

Evitando el interbloqueo

- El interbloqueo se evita impidiendo que se le dé la condición de espera circular
    - Si esta no se da, no existe el interbloqueo
- Para ello, hay que saber a priori los recursos que necesita cada proceso
    - La obtención de recursos debe ser llevada a cabo de forma que el sistema no entre en un estado inseguro
    - Algoritmo del banquero de Dijkstra
- No obstante, no es siempre posible conocer los recursos que un proceso requiere antes de su ejecución, porque depende de su ejecución

Thread Safety

Un programa, método, función o clase se dice que es thread-safe si funciona correctamente cuando se utiliza por varios hilos simultáneamente (este concepto también se aplica a estructuras de datos). El hecho de programar utilizando elementos thread-safe, no implica que el código escrito lo sea

Estructuras de datos thread-safe

- La mayor parte de tipos de propósito general ofrecidos por los lenguajes de programación no son thread-safe por cuestiones de rendimiento
- Existen dos alternativas:
    - Utilizar estructuras de datos especiales
    - O realizar la sincronización con bloqueos de objetos thread-unsafe

Implementación de Estructuras de datos thread-safe

- Una implementación trivial de estructuras de datos thread-safe es:
    - Partir de una estructura no segura
    - Crear una nueva clase con la misma interfaz
    - Realizar una composición a una instancia de la clase thread unsafe
    - Realizar un bloqueo en el cuerpo de todos los métodos utilizando un objeto privado como monitor (es mejor bloquear un objeto interno de esta estructura de datos en vez de bloquearlo a sí mismo ya que podría estar ya bloqueado en algún otro parte del código y esto podría llevarnos a un interbloqueo)
- Esta es una implementación sencilla pero muy eficiente
- En general, las operaciones concurrentes de lectura no requieren exclusión mutua de otras operaciones de lectura
- Las escrituras, no obstante, si deben bloquear a otras lecturas y escrituras

TPL y PLINQ

TPL

- Para simplificar la creación de tareas y crearlas de forma transparente, se creó esta librería
- Ofrece las siguientes ventajas:
    - Simplifica la paralización de aplicaciones
    - Escala dinámicamente el número de hilos creados en función de los números de CPUs o cores
    - Escala y gestiona dinámicamente el número de hilos creados en función del Thread Pool
    - Ofrece servicios de paralelización mediante la división de datos procesados
    - Ofrece servicios para la paralelización mediante tareas independientes
- TPL ofrece un modelo mucho más declarativo de implementar aplicaciones paralelas
- Los métodos más utilizados de esta librería son ForEach y For:
    - Ambos reciben la tarea ejecutar como un delegado action
    - ForEach crea potencialmente un hilo por cada elemento
    - For crea potencialmente un hilo a partir de un índice de comienzo y final
    - Añade una sincronización para que en la siguiente instrucción todos los hilos hayan finalizado
- TPL también añade un método Invoke para poder obtener la paralelización mediante división de tareas independiente
    - Recibe una lista variable de delegados de tipo action
    - Crea potencialmente un hilo por cada action pasado como parámetro
    - Añade una sincronización para que en la siguiente instrucción todos los hilos hayan finalizado

PLINQ

- Es una implementación paralela de LINQ
- Esta librería ofrece:
    - Paralelismo
    - De un modo declarativo
    - Transparente al número de núcleos, eligiendo dinámicamente el número de hilos concurrentes
    - Una aproximación conservadora, analizando la estructura de la consulta en tiempo de ejecución
        - Si la consulta puede obtener mejoras de rendimiento mediante paralelización se realizarán las tareas de forma concurrente
        - Ejecutándola de modo secuencial si no fuese así

Tipado dinámico y metaprogramación

Tema 5

Tipado dinámico

- El tipado dinámico es el proceso de posponer la comprobación e inferencia de tipos en tiempo de ejecución
- El tipado dinámico tiene ventajas:
  - Una mayor adaptabilidad y flexibilidad del código
  - Un mayor nivel de abstracción
- También tiene inconvenientes:
  - No tiene detección de errores de tipo en tiempo de compilación
  - El código no está optimizado

Tipado dinámico en C #

- Para posponer la comprobación e inferencia de tipos al tiempo de ejecución hay que añadir la palabra Dynamic
- El compilador no da error, contemplando que:
  - Cualquier tipo se puede convertir a Dynamic
  - Dynamic se puede convertir a cualquier tipo
  - Cualquier operación de lenguaje se puede aplicar al Dynamic
- Se comprueba en tiempo de ejecución pudiendo dar errores

Duck Typing

- Propiedades de la mayoría de lenguajes de tipo dinámico
- Proviene de la frase if it walks like a duck and quacks like a duck must be a duck
- Significa que el estado dinámico de un objeto determina qué operaciones puede realizarse con el
- Si en el momento de pasarle el mensaje m el objeto posee un método o propiedad público m apropiado, éste podrá ejecutarse

Multiple Dispatch

- Duck typing es a veces denominado late binding puesto que pospone a tiempo de ejecución la resolución del método
- Una de las limitaciones del enlace dinámico es que supone usar single dispatch:
  - Solo permite resolver un método dependiendo del tipo de un único objeto
- Cuando queremos que la resolución del método dependa de varios tipos, necesitamos multiple dispatch
- Un mecanismo de obtenerlo son los multimétodos

Reflexión

- La reflexión es la capacidad de un sistema computacional de razonar y actuar sobre sí mismo, adaptándose asimismo a condiciones cambiantes
- El dominio computacional se extiende con la representación del propio sistema, ofreciendo en tiempo de ejecución su estructura y semántica como datos computables

- Introspección: cuando se puede consultar la representación de un programa pero no modificarla
- Intercesión: cuando se permite además modificarla, desencadenando un cambio en el que éstos representan o significan

Reflexion estructural y reflexión de comportamiento

- Reflexión estructural: si la información es la estructura de un programa
- Reflexión de comportamiento: si la información de la semántica del lenguaje cambia el significado del paso de mensajes, creación de objetos, acceso...

Anotaciones o atributos

- Los atributos o anotaciones son metadatos para dar información adicional de elementos del lenguaje
- Añade información declarativa que puede ser utilizada por otra aplicación o sistema para obtener un comportamiento distinto
    - Para el resto de aplicaciones, esta información es inocua
- Se basa en el principio de Separation of Concerns
    - Dividir el código en módulos cuya funcionalidad se solape lo menor posible, evitando el entremezclado de código de distintas funcionalidades
- Los atributos son anotaciones están escritos en el propio lenguaje

Generación dinámica de código

- Es la capacidad de generar programas en tiempo de ejecución:
    - Programas que generan programas
    - El código generado de esta forma puede ser parte de la aplicación que lo generó
- Se puede utilizar para generar dinámicamente el código de nuevos métodos o clases
- El nuevo código se puede crear a partir de:
    - El propio programa: Adaptiveness
    - El usuario: Adaptability
- Es monitorizado en los lenguajes dinámicos para generar código automático de acceso a bases de datos y creación automática de vista
- También llamada programación generativa

Metaprogramación

- Es la capacidad de poder escribir programas que escriban o manipulen otros programas es un concepto muy general que puede ser utilizado en diversos escenarios
- Los siguientes elementos de lenguajes de programación están relacionados con la metaprogramación:
    - Reflexión
    - Generación dinámica de código
    - Anotaciones o atributos
    - Tipado dinámico