Mateo Rico Iglesias

Tecnologías y Paradigmas de la Programación

2º Curso de Ingeniería Informática del Software

Universidad de Oviedo

Tema 1: Lenguajes y Paradigmas de la Programación

Lenguajes 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

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 indicar en qué medida el lenguaje está más cercano al lenguaje o sistema cognitivo humano (alto nivel) o al lenguaje de la máquina (bajo nivel)
- El lenguaje ensamblador y código máquina están considerados como lenguajes de bajo nivel
- El C es considerado como 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 (DSL) suelen tener aún 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 creaación de programas meidante 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 uno 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

Pueden ser compilados o interpretados siendo su clasificación no necesariamente excluyente

- Un compilador JIT (Just-in-Time) es un tipo de compilador que traduce el código fuente o el código intermedio de un programa en código de máquina ejecutable en tiempo de ejecución, justo antes de que se necesite. A diferencia de un compilador tradicional, que realiza la traducción completa de un programa antes de su ejecución, un compilador JIT opera de manera incremental y dinámica.
- Un compilador AoT (Ahead-of-Time) es un tipo de compilador que traduce el código fuente o el código intermedio de un programa en código de máquina ejecutable antes de su ejecución, en contraste con un compilador JIT (Just-in-Time) que realiza la traducción en tiempo de ejecución.

En función de cuando se realizan las comprobaciones de tipo

Las comprobaciones de tipo pueden ser estáticas o dinámicas no siendo esta una clasificación excluyente.

- La comprobación estática de tipo ofrece un mayor nivel de seguridad y detecta los errores de tipo antes de la ejecución, pero puede requerir más esfuerzo en la declaración de tipos.
- La comprobación dinámica suele ir ligada a una mayor adaptabilidad dinámica y a una meta programación dinámica, brinda más flexibilidad, pero puede resultar en errores de tipo en tiempo de ejecución.

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 utilizan ficheros de texto para contener su código fuente y todo está representado de manera textual como su propio nombre indica.

Paradigmas de programación

Un paradigma de programación es un enfoque o estilo de programación que establece cómo deben estructurarse y organizarse los programas de computadora. Define los conceptos, principios y técnicas fundamentales que guían la resolución de problemas y la escritura de código en un lenguaje de programación determinado.

- Un 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

> Podemos ver estos paradigmas en lenguajes ya estudiados como Java (Programación orientada a objetos) o Python (Programación funcional). Aunque hay más tipos de paradigmas como la *Programación imperativa*, *Programación lógica*, *Programación estructurada* y *Programación declarativa*

Diferencias entre *Imperativo* y *declarativo*

- Programación imperativa: Se centra en las instrucciones y los cambios de estado de las variables. El control del flujo de ejecución se basa en sentencias condicionales, bucles y procedimientos.
- Programación declarativa: Se enfoca en describir el problema y no en cómo resolverlo. Los programas especifican qué resultado se desea obtener, y el sistema de ejecución

determina cómo obtenerlo.

Principales tipos de paradigmas de programación

1. **Estructurado** *(basado en procedimientos)*: paradigma imperativo en el que se utilizan 3 estructuras de control secuencial e iterativa, pudiendo agrupar estas en procedimientos o subrutinas. Este organiza el código en estructuras de control bien definidas, como secuencias, bucles y condicionales, para facilitar la legibilidad y el mantenimiento del código.
2. **Orientado a objetos**: paradigma comúnmento imperativo que utiliza los objetos. Este organiza el código en objetos que encapsulan datos y comportamiento relacionado. Los objetos interactúan entre sí a través de mensajes y heredan características de clases más generales.
3. **Funcional**: paradigma decalrativo basado en la utilización de funciones puras evitando los cambios de estado y la mutabilidad. Se enfoca en la evaluación de expresiones y la composición de funciones.
4. **Lógico**: paradigma declarativo basado en la programación de ordenadores basándose en la lógica formal y las reglas de inferencia provenientes de las matemáticas.

Paradigmas junto a sus Ventajas y Desventajas

Estructurado basado en procedimientos

Este paradigma es comúnmente llamado simplemente imperativo. Proviene de la unión de dos paradigmas históricos: el paradigma estructurado y el paradigma procedural.

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

Pardigma estructurado: paradigma en el que se utilizan 3 estructuras de control, la secuencial, condicional e iterativa.

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

VENTAJAS

1. Legibilidad: La programación estructurada utiliza estructuras de control bien definidas, como secuencias, bucles y condicionales, lo que facilita la legibilidad y comprensión del

código. Los programas estructurados son más fáciles de entender y mantener.
2. Modularidad: Los programas estructurados se dividen en procedimientos o funciones más pequeñas y autónomas. Esto permite la reutilización del código, ya que los procedimientos se pueden invocar en diferentes partes del programa.
3. Depuración más sencilla: La estructura clara y bien organizada de la programación estructurada facilita la identificación y corrección de errores en el código. Los procedimientos se pueden probar y depurar de forma independiente antes de ser integrados en el programa principal.
4. Control del flujo de ejecución: El paradigma estructurado proporciona estructuras de control como bucles y condicionales, lo que permite un control preciso del flujo de ejecución del programa. Esto facilita la implementación de lógica compleja y la toma de decisiones.

DESVENTAJAS

1. Limitaciones en la resolución de problemas complejos: La programación estructurada puede volverse limitada cuando se enfrenta a problemas más complejos y de mayor escala. Algunos problemas pueden requerir enfoques más flexibles y expresivos ofrecidos por otros paradigmas.
2. Dificultad para la gestión de cambios: Cuando un programa estructurado se vuelve muy grande, puede ser difícil realizar cambios o actualizaciones, ya que los cambios en un procedimiento pueden afectar a otros procedimientos que dependen de él.
3. Falta de reutilización específica de datos: En la programación estructurada, los datos se pasan entre procedimientos mediante parámetros, pero no hay una forma natural de encapsular datos y comportamiento relacionado en una sola entidad, como ocurre en la programación orientada a objetos.

En general, el paradigma de programación estructurado basado en procedimientos proporciona una forma organizada y legible de escribir programas, adecuada para problemas más pequeños y menos complejos. Sin embargo, puede tener limitaciones en la resolución de problemas más grandes y requiere un buen diseño modular para aprovechar al máximo sus ventajas.

Orientado a objetos (OOP)

Las características del este paradigma son las siguientes:

- Utiliza los objetos, la unión de datos y los métodos como principal abstracción definiendo programas como interacciones entre objetos.
- Se basa en la idea de modelar objetos de la vida real o introducidos en diseño mediante la condición de objetos software.

- Un programa está constituido por un conjunto de objetos que se envían mensajes entre sí considerándose por lo general un paradigma imperativo.
- Los elementos propios de este paradigma son el encapsulamiento, herencia, polimorfismo y enlace dinámico.

Este paradigma posee los siguientes modelos computacionales:

- Basados en clases:
  - Todo objeto tiene que ser instancia de una clase
  - Las clases son del tipo objeto
  - Las clases definen las estructura y comportamiento de sus instancias
- Basados en prototipos:
  - No existe el concepto de clase siendo los objetos la única entidad
  - Se define un objeto prototipo y el resto de objetos con una estructura similar se clonan a partir de ese

VENTAJAS

1. Reutilización de código: La POO fomenta la reutilización de código a través de la creación de clases y la herencia. Los objetos se pueden crear a partir de clases existentes y heredar características y comportamientos de otras clases, lo que permite ahorrar tiempo y esfuerzo al desarrollar nuevas aplicaciones.
2. Modularidad: La POO promueve el desarrollo de código modular y estructurado. Los objetos encapsulan datos y comportamiento en una sola entidad, lo que facilita su comprensión, mantenimiento y mejora independiente.
3. Flexibilidad y escalabilidad: La POO permite manejar problemas complejos de manera más eficiente y escalable. Los objetos se pueden combinar para formar jerarquías y relaciones, lo que facilita la representación de sistemas y relaciones del mundo real.
4. Modelado del mundo real: La POO permite un modelado más natural y cercano al mundo real. Los objetos y sus interacciones se pueden mapear directamente a los conceptos y entidades del dominio del problema, lo que facilita la comprensión y comunicación entre desarrolladores y expertos en el dominio.

DESVENTAJAS

1. Curva de aprendizaje inicial: La programación orientada a objetos puede requerir un mayor nivel de abstracción y comprensión conceptual en comparación con otros paradigmas. La comprensión de los conceptos fundamentales de la POO, como clases, objetos, herencia y polimorfismo, puede llevar tiempo y requerir un esfuerzo adicional.
2. Mayor complejidad: La POO puede llevar a una mayor complejidad en el diseño y desarrollo de programas. La necesidad de identificar y definir adecuadamente las clases,

relaciones y responsabilidades puede requerir una planificación y diseño cuidadosos para evitar la creación de un código innecesariamente complicado.
3. Overhead de rendimiento: En algunos casos, el enfoque orientado a objetos puede generar un pequeño overhead de rendimiento debido a la necesidad de gestionar la creación y destrucción de objetos, el uso de herencia y polimorfismo, y otras operaciones adicionales relacionadas con la gestión de objetos.
4. Mayor consumo de recursos: Los programas orientados a objetos suelen requerir más memoria y recursos del sistema debido a la creación y gestión de objetos, así como a las operaciones de herencia y polimorfismo. Sin embargo, en la mayoría de los casos, estos costos adicionales son insignificantes en comparación con los beneficios que ofrece la POO.

En general, la programación orientada a objetos ofrece numerosas ventajas, como reutilización de código, modularidad y escalabilidad, permitiendo un desarrollo más eficiente y mantenible de software. Sin embargo, también presenta desafíos asociados con la curva de aprendizaje, la complejidad y el rendimiento.

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 siendo el valor de estas dependiente en exclusiva de los valores de los parámetros por lo que si estos valores son iguales siempre devolverán el mismo resultado
- Se hace uso exhaustivo de la recursividad en lugar de la iteración como en otros paradigmas
- Los lengujes funcionales puros no utilizan ni la asignación ni la secuenciación de instrucciones
- Sus orígenes se remontan al cálculo lambda

VENTAJAS

1. Inmutabilidad y ausencia de efectos secundarios: En la programación funcional, las funciones puras no tienen efectos secundarios y no modifican el estado de las variables. Esto simplifica la comprensión y el razonamiento sobre el código, ya que no hay cambios inesperados en los datos.
2. Mayor legibilidad y mantenibilidad: La programación funcional se centra en la composición de funciones y en la separación de las preocupaciones del programa. Esto puede llevar a un código más conciso y legible, lo que facilita su mantenimiento y mejora.

3. Facilita la programación concurrente y paralela: Dado que las funciones puras no dependen del estado compartido, la programación funcional facilita la escritura de código concurrente y paralelo. Esto permite aprovechar mejor los sistemas multiprocesador y ayuda a evitar problemas como las condiciones de carrera.
4. Testing más sencillo: Las funciones puras y sin efectos secundarios son más fáciles de probar, ya que sus resultados solo dependen de los datos de entrada. Esto facilita la creación de pruebas unitarias y mejora la calidad y robustez del código.

DESVENTAJAS

1. Curva de aprendizaje: La programación funcional puede requerir un cambio de mentalidad para los programadores acostumbrados a los paradigmas imperativos. El pensamiento en términos de funciones y transformaciones puede ser inicialmente desafiante para aquellos que están más familiarizados con los enfoques tradicionales.
2. Limitaciones en ciertos escenarios: La programación funcional puede ser menos adecuada para problemas que requieren mutabilidad y cambios de estado frecuentes. Por ejemplo, en aplicaciones que involucran interacciones en tiempo real o en las que se necesita un control muy fino del estado mutable, puede ser necesario recurrir a enfoques imperativos.
3. Eficiencia: Algunas técnicas utilizadas en la programación funcional, como la recursión y la evitación de la mutabilidad, pueden requerir más recursos computacionales, como memoria y tiempo de ejecución, en comparación con enfoques imperativos optimizados.

En resumen, la programación funcional ofrece ventajas como la inmutabilidad, la ausencia de efectos secundarios y la facilidad de pruebas y mantenimiento. Sin embargo, puede requerir un período de adaptación y no es adecuada para todos los escenarios de programación. Se debe considerar cuidadosamente el contexto y los requisitos del proyecto al decidir utilizar este paradigma.

Paradigma lógico

Paradigma declarativo basado en la programación de ordenadores basándose en la lógica formal y las reglas de inferencia provenientes de las matemáticas.

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

VENTAJAS

1. Declarativo y expresivo: La programación lógica permite describir el problema y las relaciones entre los datos de manera declarativa, en lugar de tener que especificar el flujo de control detallado. Esto puede hacer que los programas sean más expresivos y

cercanos a la lógica humana, lo que facilita la comprensión y el razonamiento sobre el código.
2. Inferencia automática: En la programación lógica, el motor de inferencia se encarga de derivar las conclusiones lógicas a partir de las reglas y los hechos proporcionados. Esto puede simplificar la resolución de problemas complejos, ya que no es necesario especificar el algoritmo de búsqueda explícitamente.
3. Modularidad y reutilización: La programación lógica fomenta la modularidad y la reutilización de código a través de la definición de reglas y hechos independientes. Esto permite construir programas a partir de componentes lógicos bien definidos y facilita la evolución y el mantenimiento del código.

DESVENTAJAS

1. Eficiencia y rendimiento: Aunque la programación lógica tiene ventajas desde un punto de vista declarativo, puede ser menos eficiente en términos de tiempo de ejecución y uso de recursos en comparación con otros paradigmas. El motor de inferencia puede requerir una gran cantidad de procesamiento para encontrar soluciones a problemas complejos.
2. Dificultad en el control del flujo de ejecución: A diferencia de los paradigmas imperativos o procedurales, en la programación lógica no se especifica directamente el flujo de ejecución. Esto puede dificultar el control preciso del orden de ejecución y puede llevar a resultados inesperados si no se tienen en cuenta las reglas y hechos correctamente.
3. Expresividad limitada en ciertos problemas: Aunque la programación lógica es efectiva para problemas que se pueden representar de manera lógica y que se adaptan bien a las reglas y la inferencia, puede resultar menos adecuada para problemas que no se prestan a una representación lógica clara. En tales casos, otros paradigmas pueden ser más apropiados.

En resumen, la programación lógica tiene ventajas en términos de declaratividad, inferencia automática y modularidad. Sin embargo, puede tener desventajas en cuanto a eficiencia, control del flujo de ejecución y expresividad limitada en ciertos problemas. Es importante considerar el tipo de problema y los requisitos específicos del proyecto al decidir utilizar la programación lógica como enfoque.

Tema 2: Elementos del paradigma orientado a objetos en 'C#'

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 que interaccionan entre sí

Abstracción

Esta expresa las características esenciales de un objeto, las cuales distinguen al objeto de los demás.

Encapsulamiento

Es el 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 que parte de una abstracción está disponible para el resto de la aplicación y que parte es solo interna de la propia abstracción.

Para este propósito los lenguajes de programación ofrecen diversos niveles de ocultación para sus miembros. Por lo general cada objeto estaría aislado del exterior y expondría una interfaz de cara a otros objetos en la que se especifica como deben estos objetos interactuar con el objeto aislado.

Propiedades

Resumido

C# ofrece el concepto de propiedad para acceder al estado de los objetos como si del atributo se tratase, obteniendo los beneficos del encapsulamiento. Estas propiedades pueden ser tanto de lectura (get) como de escritura (set) y pueden catalogarse con cualquier nivel de ocultación como también ser abstractas o sobreescribirse.

Detallado

En el lenguaje de programación C#, las propiedades son miembros de una clase que encapsulan los campos o variables de instancia y proporcionan una forma de acceder y modificar sus valores. En lugar de acceder directamente a los campos, se utilizan métodos de acceso llamados "getters" y "setters" para leer y escribir los valores de las propiedades.

Las propiedades son miembros de una clase que encapsulan los campos o variables de instancia y proporcionan una forma de acceder y modificar sus valores. En lugar de acceder directamente a los campos, se utilizan métodos de acceso llamados "getters" y "setters" para leer y escribir los valores de las propiedades.

Las propiedades en C# se definen mediante un par de métodos de acceso: el método `get` y el método `set`. El método `get` se utiliza para recuperar el valor de una propiedad, mientras que el método `set` se utiliza para asignar un valor a la propiedad. Estos métodos se definen dentro de la declaración de la propiedad.

```
public class Persona
{
    private string nombre;

    public string Nombre
    {
        get { return nombre; }
        set { nombre = value; }
    }
}
```

En este ejemplo, la clase `Persona` tiene una propiedad llamada `Nombre` que encapsula el campo privado `nombre`. El método `get` devuelve el valor del campo `nombre`, mientras que el método `set` asigna un nuevo valor al campo `nombre`.

Puedes acceder y modificar el valor de la propiedad `Nombre` utilizando la siguiente sintaxis:

```
Persona persona = new Persona();
persona.Nombre = "Juan";
Console.WriteLine(persona.Nombre); // Imprime "Juan"
```

En este caso, la propiedad `Nombre` se utiliza como si fuera un campo público, pero en realidad, se están invocando los métodos de acceso `get` y `set` internamente.

Las propiedades en C# permiten un mejor encapsulamiento de los datos y ofrecen un control más preciso sobre el acceso y la modificación de los campos de una clase. También proporcionan la capacidad de agregar lógica adicional dentro de los métodos de acceso, como realizar validaciones o cálculos adicionales.

Además de las propiedades de instancia, C# también admite propiedades estáticas, que se asocian con toda la clase en lugar de una instancia específica.

Modularidad

Pripiedades que permiten subdividir una aplicación en partes más pequeñas, siendo cada una de estas tan independiente como sea posible. Cada una de estas partes/módulos ha de poder ser compilada por separado para ser utilizada en diversos programas.

Hay diversos elementos que pueden constituir un módulo:

- Funciones y métodos
- Clases y tipos
- Espacios de nombres y paquetes
- Componentes

Acoplamiento y cohesión

El ***acoplamiento*** indica el nivel de interdependencia entre módulos. Por su parte, la ***cohesión***, indica el nivel de uniformidad y relación que existe entre las distintas responsabilidades de un módulo.

En el desarrollo de software el bajo acoplamiento y la elevada cohesión favorecen 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 uno de los siguientes puntos:

- El número de parámetros
- El tipo de algunos de sus parámetros
- El paso de algunos de sus parámetros (valor, ref, out)
  En Java el último punto no se puede aplicar ya que el lenguaje no cuenta con esta característica.

Sobrecarga de operadores

La sobrecarga de operadores permite modificar la semántica de los operadores del lenguaje. En C# se ofrece esta característica pudiendo modificar operadores como '++', '--', '[ ]', ... A pesar de estar disponible en C# no es una característica muy utilizada.

Herencia

La herencia es un concepto fundamental en la programación orientada a objetos (POO) que permite que una clase herede características y comportamientos de otra clase. En la herencia, una clase llamada "clase derivada" o "subclase" puede extender y reutilizar los miembros (atributos y métodos) de otra clase llamada "clase base" o "superclase".

La herencia se basa en la idea de que ciertas clases tienen una relación "es un" con otras clases. Por ejemplo, podríamos tener una clase base llamada "Animal" y clases derivadas

como "Perro", "Gato" y "Ave". En este caso, podemos decir que un perro es un animal, un gato es un animal y un ave es un animal. La clase base "Animal" contiene las características y comportamientos comunes a todos los animales, mientras que las clases derivadas agregan características y comportamientos específicos de cada tipo de animal.

```
class ClaseDerivada : ClaseBase
{
    // Miembros adicionales de la clase derivada
}
```

La clase derivada hereda todos los miembros (atributos y métodos) públicos y protegidos de la clase base. Esto significa que la clase derivada puede acceder y utilizar los miembros heredados como si fueran propios, y también puede agregar nuevos miembros o modificar los existentes.

La herencia permite varios beneficios, entre ellos:

1. **Reutilización de código**: Al heredar de una clase base, puedes aprovechar los miembros existentes y evitar la duplicación de código. Esto promueve la reutilización y el mantenimiento eficiente del código.
2. **Extensibilidad**: Puedes agregar nuevos miembros a la clase derivada para ampliar y especializar su funcionalidad. Esto permite modelar relaciones jerárquicas y comportamientos específicos.
3. **Polimorfismo**: Las clases derivadas pueden reemplazar los métodos heredados de la clase base con sus propias implementaciones, lo que permite el polimorfismo. Esto significa que una instancia de la clase derivada se puede tratar como una instancia de la clase base, lo que facilita la flexibilidad y modularidad en el diseño de programas.

Es importante tener en cuenta que la herencia debe usarse con cuidado y seguir los principios de diseño adecuados. Una jerarquía de herencia mal diseñada o demasiado compleja puede llevar a problemas de mantenimiento y dificultades en la comprensión del código.

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, desde lo más específico hasta los más generalizado, 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

Este tipo de herencia produce dos conflictos principales:

- *Coincidencia de nombres*: se produce cuando se hereda de dos o más clases un miembro con igual identificador lo que produciría una ambigüedad en su acceso.
- Herencia repetida: se produce cuando se hereda más de una vez de una clase por distintos caminos.

Interfaces

En programación orientada a objetos, una interfaz es una especificación de un conjunto de métodos que una clase debe implementar. Representa un contrato que define qué operaciones puede realizar un objeto sin especificar cómo se implementan dichas operaciones.

En C#, una interfaz se define utilizando la palabra clave `interface`. Una interfaz declara los métodos, propiedades, eventos y otros miembros que deben ser implementados por cualquier clase que la utilice. Una clase puede implementar múltiples interfaces, lo que permite lograr una mayor flexibilidad y modularidad en el diseño del código.

```
public interface IReproductorMusica
{
    void Reproducir();
    void Pausar();
    void Detener();
}
```

En este ejemplo, se define una interfaz llamada `IReproductorMusica` que declara tres métodos: `Reproducir()`, `Pausar()` y `Detener()`. Cualquier clase que implemente esta interfaz deberá proporcionar una implementación para estos métodos.

Una vez que se define una interfaz, se puede implementar en una clase utilizando la palabra clave `implements` en C#. Aquí tienes un ejemplo de cómo se implementaría la interfaz `IReproductorMusica` en una clase `ReproductorMp3`:

```
public class ReproductorMp3 : IReproductorMusica
{
    public void Reproducir()
    {
        // Implementación del método Reproducir
    }

    public void Pausar()
    {
        // Implementación del método Pausar
    }

    public void Detener()
    {
        // Implementación del método Detener
    }
}
```

Al implementar una interfaz, la clase debe proporcionar una implementación para todos los miembros declarados en la interfaz. Esto garantiza que la clase cumpla con el contrato especificado por la interfaz y pueda utilizarse de manera intercambiable con otras clases que implementen la misma interfaz.

Las interfaces son útiles en varios escenarios, incluyendo:

1. **Contratos de comportamiento**: Las interfaces definen un contrato común que las clases deben cumplir, lo que permite la interoperabilidad y el diseño basado en contratos.
2. **Polimorfismo**: Las interfaces permiten el polimorfismo, lo que significa que una instancia de una clase que implementa una interfaz puede ser tratada como una instancia de la interfaz. Esto facilita la modularidad y la extensibilidad del código.
3. **Separación de preocupaciones**: Las interfaces ayudan a separar la definición de un comportamiento de su implementación específica. Esto promueve una mejor organización y estructura en el código.

En resumen, una interfaz en C# es una especificación de métodos, propiedades u otros miembros que una clase debe implementar. Proporciona un contrato común que define el comportamiento esperado sin especificar la implementación concreta de los métodos.

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 la ejecución
3. Extensible: en función del uso de una abstracción un error puede
    1. Transformarse en otros errores
    2. 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. Estos no se deben utilizar para detectar errores en tiempo de ejecución ya que no son ni reutilizables ni extensibles. Los asertos se podrán utilizar para detectar aquellas situaciones que nunca deben ocurrir y en caso de que esto pase se trataría de un error que debemos solucionar. Estos asertos se deberan eliminar una vez solucionado el error que daba sentido a su utilización.

Genericidad

La genericidad es la propiedad que permite construir abstracciones modelo para otras abstracciones. Esta ofrece dos principales beneficios:

- Una mayor robusted
- Un mayor rendimiento
  La genericidad acotada permite hacer que los tipos genéricos sean más específicos.

Tema 3: Fundamentos del paradigma funcional

Paradigma funcional

Paradigma declarativo basado en la utilización de funciones que manejan datos inmutables, es decir que nunca se modifican y que en caso de querer cambiarlos se llama a una función que devuelve una copia del dato modificado pero sin cambiar el original.

Un programa se puede definir como un conjunto de funciones invocándose entre si. Estas funciones no generan efectos secundarios ya que el valor de una expresión dependerá exclusivamente de los parametros de la misma.

Cálculo lambda

Es un sistema formal basado en la definición de funciones y su aplicación, este se considera como el lenguaje universal más pequeño en computación, considerándose universal debido a que cualquier función computable puede ser expresada y evaluada utilizándolo.

Expresiones lambda

En el cálculo lambda, una expresión se define como una abstracción en la que x es una variable (parámetro) y M es una expresión lambda (cuerpo de función). También se define como una aplicación MN donde M y N son expresiones lambda.

Ejemplo

- La función identidad f(x)=x puede representarse como λx.x
- La función identidad f(x)=x+x puede representarse como λx.x+x (x+x no sería una expresión lambda pero el ejemplo trata de buscar la mayor simplicidad)

Aplicación (Reducción β)

La aplicación de una función representaría su invocación y se puede definir del siguiente modo, `(λx.M)N → M[x:=N] (o M[N/x])` siendo x una variable y M y N expresiones lambda. `M[x:=N]` representa M donde todas las apariciones de x son sustituidas por N.

Esta sustitución es lo que se denominaría reducción β, algunos ejemplos de su aplicación serían los siguientes:

- (λx.x+x)3 →3+3
- (λx.x) λy.y2 → *λy.y2*

Teorema de Church-Rosser

En algunos terminos 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 lo que los paréntesis se usan de manera general para delimitar términos lambda y no indican preferencias.

Variables libres y ligadas

En la abstracción λx.xy se dice que la variable ***x*** está ligada y la ***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 en la que todas las apariciones de una variable ligada en una misma abstracción se pueden renombrar a una nueva variable.

```
Ej.
(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 siendo esta en el cálculo lambda una correspondencia directa.

Esto implica que:

- Exite una correspondencia entre tipos y proposiciones
- Existe una correspondencia entre programas y evidencias que demuestran las 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 traducción directa con la lógica

Existen asistentes de demostradores que permiten demostrar propiedades de programas. Algunas de sus características son:

- Se basa en leguajes 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 de primer orden

El paradigma funcional identifica las funciones como entidades de primer orden, igual que el resto de valores (funciones de primer orden). Por esto las funciones se consideran como 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 de orden superior si recibe alguna función como parámetro o retorna una función como resultado.

Funciones de orden superior

Una función de orden superior, en el contexto de la programación funcional, es una función que puede recibir otras funciones como argumentos y/o devolver una función como resultado. En otras palabras, trata a las funciones como ciudadanos de primera clase, permitiendo su manipulación y uso como cualquier otro tipo de dato.

En lenguajes de programación que admiten funciones de orden superior, las funciones se pueden tratar como valores y se pueden pasar como argumentos a otras funciones. Esto permite una mayor flexibilidad y expresividad en el diseño de programas, ya que las funciones pueden ser abstractas y parametrizadas, lo que permite crear comportamientos genéricos y reutilizables.

Un ejemplo de está sería por ejemplo la función `λf.( λx.f(fx))` que vimos anteriormente.

Delegados

En C# las funciones son entidades de primer orden gracias a los delegados. Estos constituyen un tipo que representa un método de instancia o de clase representando las variables de este tipo un modo de referenciar un método.