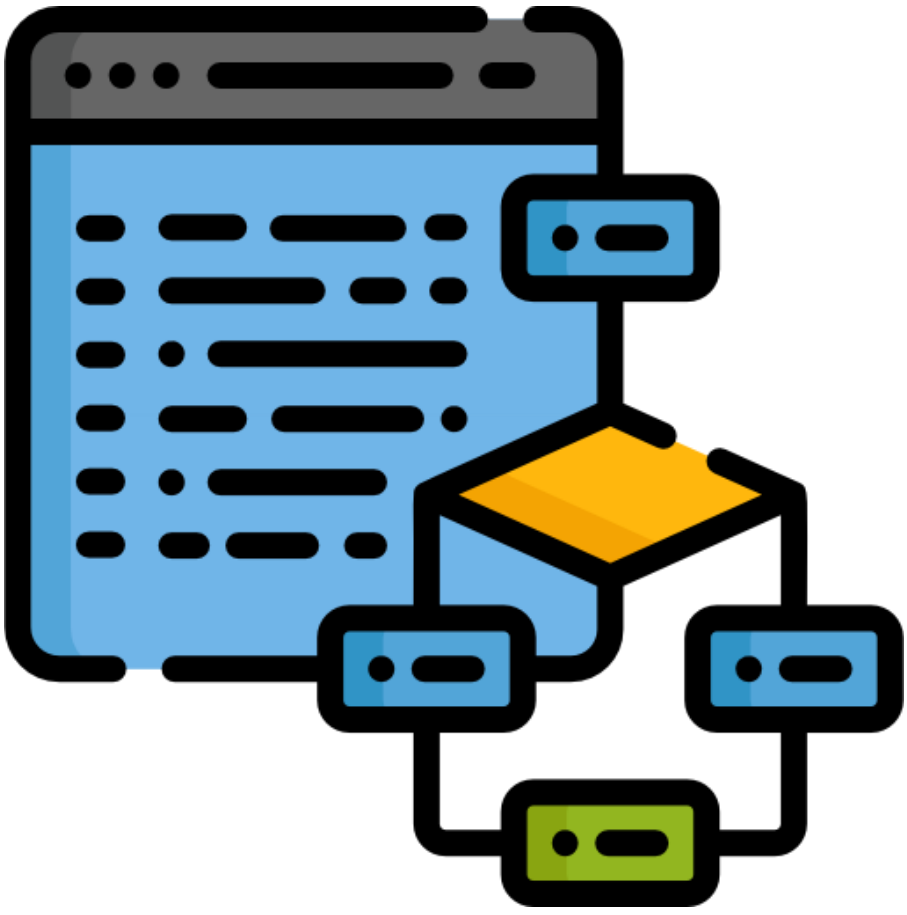
PROGRAMACIÓN ORIENTADA A OBJETOS



2º GREI USC

TEMA 1. Encapsulación. Tipos de datos, clases y objetos.

1. Tipos de datos

Un programa se puede entender como una serie de instrucciones que operan sobre un conjunto de datos (entrada) y generan otro conjunto de datos (salida). Todos los lenguajes de programación utilizados en la actualidad tienen instrucciones de selección (tipo if) y de repetición (tipo while o for), de forma que es posible crear cualquier programa (*teorema de Jacopini*). Los tipos de datos disponibles en un lenguaje de programación limitan enormemente las posibilidades a la hora de crear programas si no es posible representar un tipo de datos concreto, no se podrá operar sobre ellos.

CLASE = TIPO DE DATOS

Ejemplo: si un lenguaje de programación no dispone de datos de tipo decimal, no podrá crear un programa que calcule el área de un círculo.

Tipos de datos C vs Python:

Tipo	Rango
char	-127 a 127
unsigned char	0 a 255
signed char	-127 a 127
int	-32767 a 32767
unsigned int	0 a 65535
signed int	-32767 a 32767
short int	-32767 a 32767
unsigned short int	0 a 65535
lont int	-2147483647 a 214783647
signed long int	-2147483647 a 214783647
unsigned long int	0 a 4294967295
float	6 dígitos de precisión
double	10 dígitos de precisión
long double	10 dígitos de precisión

Tipo	Notas
int	Número entero
float	Coma flotante
bool	Valor verdadero o falso
str	Inmutable
list	Mutable
tuple	Inmutable
set	Mutable, sin orden, sin duplicados
frozenset	Inmutable, sin orden, sin duplicados

¿Qué tipos de datos son los que usamos a la hora de desarrollar un programa?

Los tipos de datos que necesitamos en un programa no son tipos de datos que están disponibles en c, c++, java…, tal y como vemos en el caso de este juego:

> El juego tiene lugar sobre un tablero en el que se representa un **mapa** del mundo. En el **mapa** se dispondrán 6 **continentes** que se identificarán con un **color**: Asia (CYAN), África (VERDE), Europa (AMARILLO), América del Norte (VIOLETA), América del Sur (ROJO) y Australia (AZUL).
>
> Las **casillas** son los **países** de los que consta cada **continente**. Es decir, los **países** se representarán como **casillas** cuadradas o rectangulares para facilitar su pintado.
>
> Las **casillas** en NEGRO se considerarán océanos o mares.

2. Encapsulación

Una de las principales características de la Programación Orientada a objetos es la **encapsulación**.

- Se asegura la integridad de los datos, limitando los trozos de código del programa que pueden acceder a unos datos dados.
- Idealmente no existe ningún trozo de código que pueda acceder a todos los datos del programa.
- Hace uso del **principio de ocultación**, en virtud del cual solo un trozo de código puede "ver" un conjunto de datos dado.

3. Clases

En Programación Orientada a Objetos los tipos de datos son las entidades que se quieren representar en el programa

- Si necesitamos representar la edad de una persona, esa entidad será un entero
- Si necesitamos representar un continente, entonces la entidad será el continente y tendrá una serie de atributos (o variables) que lo definen y que, a su vez, tienen un tipo de dato asociado.
    - El color del continente es una cadena de texto
    - Los países del continente son un array de tipo País

En Programación Orientada a Objetos estas entidades del programa, es decir, los tipos de datos, se conceptualizan como clases.

En Programación Orientada a Objetos todos los tipos de datos son clases y los valores concretos de esos tipos de datos, es decir, de las clases, se denominan objetos:

- Asia es un objeto que tiene como tipo de dato Continente.
- Venezuela es un objeto que tiene como tipo de dato País.
- Amarillo es un objeto que tiene como tipo de datos una cadena de texto.

CLASES EN JAVA

Java es un lenguaje de programación que está basado en el paradigma de la Programación Orientada a Objetos que distingue entre dos especies de tipos de datos:

- Tipos de datos primitivos, que son los tipos de datos que están vinculados a la representación de los datos en el computador. Son los mismos tipos de dato que los del lenguaje C.
- Clases, que son tipos de datos que están vinculados a las entidades de alto nivel del programa y, por exclusión, son tipos de datos que no se pueden representar de forma directa en el computador.

Con carácter general, las clases contienen dos tipos de código:

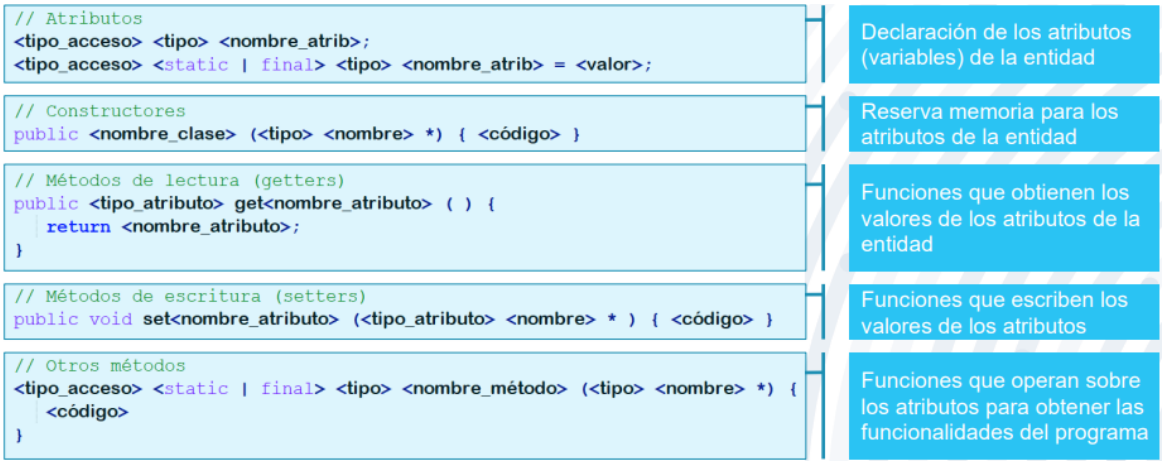
- Las variables o atributos que son las características que definen la entidad representada por la clase

Ejemplo: el nombre del continente, el conjunto de países que forman parte del continente, el conjunto de continentes con los que limita.

- Las funciones o métodos que acceden a los atributos para permitir su lectura y escritura, y para proporcionar la funcionalidad requerida en el programa

  Ejemplo: la función que obtiene el conjunto de países que son frontera de un continente.

La idea para resolver el problema de la integridad de los datos es que las únicas funcionas que pueden acceder a los atributos, tanto para leerlos como para leerlos como para escribir sobre ellos, son las funciones de la clase



En Java de dispone de **cuatro tipos de acceso**, que se pueden aplicar tanto a los atributos como a los métodos:

- Público (public)
  - Los métodos públicos pueden ser invocados desde cualquier método de cualquier clase del programa.
  - Los atributos públicos, pueden ser leídos o escritos desde cualquier método de cualquier clase del programa.
- Privado (private)
  - Los métodos privados, solamente pueden ser invocados desde cualquier método de la clase a la que pertenecen.
  - Los atributos privados, solamente se pueden leer o escribir desde los métodos de la clase a la que pertenecen.

  Para leer y escribir los atributos los atributos privados de una clase se usan dos tipos de métodos: getters y setters, respectivamente.

- Acceso a paquete
  - Los métodos de acceso a paquete, solamente pueden invocarlos los métodos de las clases que pertenecen al mismo paquete de la clase en la que se encuentra el método. Se comportan como privados para cualquier otro método.
  - Los atributos de acceso a paquete, pueden ser leídos y escritos directamente por los métodos de las clases que pertenecen al mismo paquete de la clase do los atributos. Se comportan como privados para cualquier otro método. (*diapositiva 21*)

- Acceso protegido (protected)
    - Los métodos protegidos, pueden ser invocados por los métodos de las clases que pertenecen al mismo paquete de la clase y por los métodos de las subclases de la clase.
    - Se introducen para facilitar la herencia de atributos entre clases que se encuentran en paquetes diferentes.

GETTERS

Los getters son métodos que devuelven el valor que tienen los atributos en cada momento.

- Cada atributo de la clase tiene un único getter
- El nombre del getter es get<nombre_atributo>, donde la primera de <nombre_atributo> es en mayúscula.
- Típicamente los getters únicamente contienen el código asociado a la devolución del valor del atributo (return <valor_atributo>)

SETTERS

Los setters son métodos que escriben el valor de los atributos

- Cada atributo de la clase tiene un único setter
- El nombre del getter es set<nombre_atributo>, donde la primera de <nombre_atributo> es en mayúscula.
- Típicamente un setter tiene código sobre las condiciones que debe de cumplir el argumento para que sea un valor correcto del atributo, manteniendo así la integridad de los datos.

CONSTRUCTORES

La declaración de un atributo de una clase tiene distintos efectos en función de su tipo de dato.

- Si el atributo es un tipo de dato primitivo
- Si el atributo es de tipo clase

De la misma forma, al declarar una variable que tiene como tipo de dato una clase, es decir, al declarar un objeto, no se reserva automáticamente memoria para dicho objeto, por lo que si una variable no se inicializa o no se reserva memoria, se genera un error de compilación.

Un constructor es un método/función que se invoca para:

- Reservar memoria para un objeto de la clase
- Reservar memoria para los atributos de dicho objeto
- Asignar valores iniciales a los atributos del objeto

Un constructor se invoca una única vez para cada objeto. En realidad cuando se reserva memoria para un objeto, el nombre del objeto es una referencia a la posición de memoria en la que se almacenan los datos del objeto. Una vez se ha creado la referencia y se ha invocado al constructor para reservar memoria, si se vuelve a invocar al constructor, se apuntará a una nueva posición de memoria.

MÉTODOS FUNCIONALES

Una vez han sido definidos los atributos que caracterizan a una clase, se identifican y codifican los métodos que pertenecen a dicha clase y que realizan las operaciones que necesitan otras clases para implementar la funcionalidad del programa.

Los métodos funcionales acceden directamente a los valores de los atributos. Tiene sentido crear este tipo de métodos en una clase cuando los datos necesarios para realizar las operaciones son los valores de los atributos de dicha clase. Los métodos se invocan desde los objetos de las clases con el operador ¨.¨.

Un método o constructor de una clase puede tener varias implementaciones, lo que otorga una mayor flexibilidad a la hora de usar la clase en diferentes contextos.

No hay una regla fija que indique cuántas implementaciones diferentes debe tener un método en una misma clase y en la mayoría de los casos depende del contexto de invocación de los métodos:

- El objetivo del método tiene que ser el mismo en todas las implementaciones.
- Se desaconseja el uso de condiciones sobre los argumentos con el fin de seleccionar el trozo de código que se deberá ejecutar en cada caso.

La sobrecarga de constructores es mucho más habitual, ya que tienen diferentes argumentos en función de la disponibilidad de los valores de los atributos a la hora de crear la clase.

ALMACENAMIENTO DE DATOS

Dependiendo del tipo de dato y del lugar del programa en el que se definen, los datos se almacenan en zonas de memoria diferentes.

Pila (Stack): zona de la memoria a la que el procesador tiene acceso directo a través de un puntero de pila, y en la que la lectura y escritura es rápida y eficiente.

El compilador debe conocer con antelación cuanta memoria se necesita reservar en la pila, ya que debe de mover el puntero a lo largo de la pila para acceder a los datos.

Las variables existen en la pila durante la ejecución del método que las ha creado, de modo que cuando se finaliza su ejecución, se eliminan automáticamente de la memoria.

Los datos que se almacenan en la pila siempre deben tener un tamaño conocido

- Todo el código correspondiente a los métodos (call stack)
- Todos los fatos de tipo primitivo usados durante la ejecución de los métodos
- Las referencias a los objetos creados en el programa.

Montón (Heap): zona de la memoria en la que el procesador no necesita conocer que cantidad de datos se deben reservar y cuánto tiempo van a estar disponibles esos datos.

- Almacena a los objetos creados durante la ejecución del programa.
- La gestión del montón corre a cargo del recolector de basura, ya que los datos no eliminan automáticamente cuando dejan de ser necesarios.

- El rendimiento de un programa en Java está muy condicionado por la gestión eficiente del montón, o lo que es lo mismo, por el rendimiento del recolector de basura.

INVOCACIÓN DE MÉTODOS

Una clase no ocupa memoria. Las instrucciones de los métodos se cargan en memoria, en la pila, cuando se crea un objeto de la clase a la que pertenecen dichos métodos. A partir del momento en el que se crea el objeto, o instancia, es posible acceder a todos los métodos a través del operador ¨.¨.

Cuando en el método se hace referencia a un atributo de la clase a la que se pertenece, este atributo tomara los valores asociados al objeto que hace uso del operador ¨.¨

JAVA: Método toString

toString es un método que devuelve la representación en texto de un objeto de una clase. toString es un método que tienen todas las clases de Java, tanto las propias de la distribución Java como las que se crean en el desarrollo de los programas. Las clases heredan la implementación de toString que tiene la clase Object, que es la clase que está en el nivel más alto de la jerarquía Java.

La implementación que se hereda de Object es muy genérica y realmente no aporta información sobre el contenido del objeto, ya que solamente se compone del nombre de la clase a la que pertenece el objeto y de un código hash que lo identifica.

Es necesario reimplementar el método toString para devolver una representación en texto con aquellas características que se consideren más relevantes para describir el objeto.

TEMA 2. Clases y tipos de datos

1. Tipos de datos primitivos

Los tipos de datos primitivos son tipos de datos predefinidos que tienen una correspondencia directa con los tipos de datos de otros lenguajes basados en procedimientos, como C.

En cuanto a sus características, tienen un tamaño fijo, tienen una correspondencia directa con los tipos de datos que es posible representar en un computador, en el momento de su declaración se realiza automáticamente la reserva de memoria para la variable y no tienen métodos.

Los tipos de datos primitivos no son clases, puesto que no encapsulan ni datos ni métodos que acceden y modifican dichos datos, si no que están directamente vinculados a los valores de las variables.

Para hacer consistente el esquema de tipos de datos de Java con la programación orientada a objetos, para cada tipo de datos primitivos se han definido wrappers que envuelven los valores de los tipos en diferentes tipos de objetos en función de los datos primitivos.

Estos wrappers son clases de Java que encapsulan el valor del tipo de dato y proporcionan una serie de métodos que facilitan operaciones comunes sobre los datos.

Los métodos de los wrappers de los tipos primitivos permiten obtener el valor del dato; convertir el dato a una cadena de texto, y viceversa; o convertir el dato a otros tipos de datos primitivos.

2. Wrappers

El uso de wrappers proporciona una gran versatilidad en el manejo de datos en el lenguaje Java, pero genera código que es mucho más difícil de entender que el que genera el uso de datos primitivos.

El uso de autoboxing y unboxing resuelve este problema, ya que permite tratar un wrapper como si fuese un tipo de dato primitivo, y viceversa; pudiendo elegir en cada momento el comportamiento que se desee.

⇒ **Autoboxing:** convierte un tipo de dato primitivo a un objeto de la correspondiente clase wrapper. Se aplica autoboxing cuando (a) un método que tiene como argumento un tipo wrapper, recibe un valor de tipo primitivo; o (b) a un objeto de tipo wrapper se le asigna un valor de tipo primitivo.

⇒ **Unboxing:** convierte un objeto de una clase wrapper al tipo de dato primitivo correspondiente. Se aplica unboxing cuando (a) un método que tiene como argumento un tipo primitivo recibe un objeto de una clase wrapper; o (b) a un tipo de dato primitivo se le asigna un objeto de tipo wrapper.

3. Referencias

En Java los nombres de los objetos son referencias a la posición de memoria que está reservada para dichos objetos.

Este uso de referencias tiene importantes implicaciones. Cuando se asigna un objeto obj_a a otro objeto obj_b, en realidad no se realiza una copia de la memoria que ocupa obj_b en la posición de memoria que ocupa obj_a. En esta situación se está realizando una asignación de la referencia del objeto obj_b a la referencia al obj_b, o lo que es lo mismo, ambas referencias apuntan a la misma posición de memoria. La memoria del objeto obj_a ya no está disponible.

La asignación de referencias entre dos objetos se denomina **aliasing** y es una de las principales características no solo de Java, si no de la mayoría de lenguajes orientados a objetos.

El uso de aliasing evita la encapsulación de los datos, ya que permite modificar el valor de los atributos desde métodos que no pertenecen a las clases que contienen dichos atributos.

Evitar aliasing en los lenguajes de programación orientada a objetos reduciría enormemente su rendimiento, puesto que supondría que la asignación entre dos objetos sería una copia completa de una zona de memoria a otra zona de memoria del montón.

Desde un punto de vista práctico, es muy difícil el desarrollo de programas en los que no se haga uso de aliasing en alguna parte del código. No siempre es adecuado evitar aliasing y es necesario seleccionar aquellas partes del código en las que se debe evitar aliasing.

La única forma de evitar aliasing es introducir manualmente código que genere una nueva referencia del objeto, es decir, reserve memoria, y copie los atributos del objeto generando con nuevas referencias cuando sea necesario. En vez de que un método devuelva una referencia a un atributo, deberá crear y devolver un objeto del mismo tipo del atributo que ocupa una posición de memoria diferente. Además, en vez de que un método acepte como entrada la referencia de un objeto, deberá crear un objeto del mismo tipo del argumento para que ocupe una posición de memoria diferente. Para facilitar estas operaciones, Java proporciona el método clone, que puede implementarse en todos los objetos.

El método clone está pensado para generar una copia exacta de un objeto, almacenando esa copia en una posición de memoria diferente de la que ocupa dicho objeto. El programador debe implementar explícitamente el método para cada clase de cuyos objetos se desea realizar una copia. No es práctico implementar el método clone para todas las clases de un programa, sobre todo si se trata de realizar copias profundas. En una copia profunda es necesario reservar memoria para todos los atributos de la clase, incluyendo todos los elementos de un conjunto de datos.

CLASE STRING

String es una clase de Java, pero no se comporta como el resto de las clases en lo que respecta a la asignación entre objetos. Cuando se realiza una asignación entre una cadena de texto CT_A y una cadena de texto CT_B , en realidad, se está reservando memoria para la primera cadena de texto, CT_A , y se le asigna el valor de la segunda cadena de texto, CT_B .En realidad una cadena de texto es un objeto inmutable, es decir, una vez se ha reservado memoria y se le asigna un valor dado, no se puede modificar.

Los objetos que son cadenas de texto se pueden crear de dos formas diferentes:

- Directamente: asignando una cadena de texto al objeto, en cuyo caso se almacenan en una zona del montón llamada String Pool, de modo que cada vez que se asigna la misma cadena de texto, se apunta a la dirección que la contiene (str1 y str2).
- Indirectamente: el objeto se crea usando un constructor de String, en cuyo caso se almacenan en el montón, pero fuera del String Pool, aunque tenga el mismo valor que una cadena previa.

El uso indiscriminado de cadenas de texto puede llevar a una merma importante en el rendimiento de un programa.

4. Identidad de objetos**: Método equals**

¿Cuándo se puede considerar que dos objetos son iguales?

- Opción 1: dos objetos son iguales si ocupan la misma posición de memoria.
  - Esta condición es muy restrictiva, ya que en realidad no están comparando dos objetos, si no que se comparan dos referencias a un mismo objeto.
- Opción 2: dos objetos son iguales si son del mismo tipo y si los valores de todos los atributos de los objetos son también iguales
  - Esta condición obliga a que los dos objetos tengan los mismos valores de los atributos en el momento de la comparación y que los atributos sean inmutables.

Equals es un método que indica si un objeto, que es el argumento del método, es igual al objeto que se invoca en equals. Este es un método que tienen todas las clases de Java, tanto las propias de la distribución de Java como las que se crean en el desarrollo de programas. Las clases heredan la implementación de equals que tiene la clase Object, que es la clase que está en el nivel más alto de la jerarquía de Java.

La implementación que se hereda de Object sigue la opción 1, es decir, compara las referencias de los dos objetos, ya que lo único que se conoce de cualquier objeto es su referencia. Es necesario reimplementar el método equals identificando los atributos que son inmutables una vez que se reserva memoria para el objeto, de modo que dos objetos son iguales si los atributos inmutables tienen los mismos valores. Típicamente los objetos inmutables son cadenas de texto o wrappers asociados a tipos primitivos.

TEMA 3. Conjuntos de datos: Arrays, colecciones, listas, conjuntos y mapas

1. Conjuntos de datos

Una buena parte de la programación consiste en manejar conjuntos de datos sobre los que se realizan operaciones CRUD de búsqueda, inserción, modificación y borrado. Típicamente la mayoría de los lenguajes de programación soportan la representación de conjuntos de datos a través del concepto de array.

Los arrays tienen muchas limitaciones a la hora de acceder a los datos, tanto en la lectura como en la escritura. Se han propuesto una buena cantidad de formas de representar conjunto de datos que intentan dar respuesta a esas limitaciones. Java soporta directamente varios tipos de conjuntos de datos.

Arrays	Elementos accesibles a través de un índice
Colección	Grupo de objetos
Iterador	Objeto que itera sobre una colección
Lista	Colección ordenada de objetos
Conjunto	Colección sin objetos duplicados
Mapa	Objeto que relaciona claves con valores

2. Arrays

El concepto de array en Java es el mismo que en los lenguajes procedimentales, es decir, un conjunto de elementos del mismo tipo que ocupan posiciones consecutivas de memoria para facilitar su acceso de forma sencilla a través de índice.

En Java un array de datos es un objeto. Se debe usar new para reservar memoria, no es suficiente con declararlo. Hereda todos los métodos de la clase Object y, además, tiene como atributo público la longitud del array (length).

Son las únicas estructuras que permiten almacenar tipos de datos primitivos. El acceso a los elementos de un array en Java es muy parecido al de los lenguajes procedimentales. Se accede a los elementos a través de un índice, siendo el índice del primer elemento 0, y el del último elemento length-1. Si se intenta acceder a una posición > length-1 se genera un error ArrayIndexOutOfBoundsException.

Los arrays tienen importantes limitaciones desde el punto de vista de las operaciones que se pueden realizar sobre los datos; tienen un tamaño fijo, que no se puede modificar y no es posible borrar datos cuando son tipos primitivos.

3. Colecciones, listas, iteradores y mapas

Todos los tipos de conjuntos de datos son mapas (Map), colecciones (Collection) o iteradores (Iterator). Los otros tipos de conjuntos de datos tienen características que particularizan el tipo que hereda. Los ArrayList y los HashMap son tipos de listas y de mapas, respectivamente, que se usan con mucha frecuencia en los programas de Java.

COLECCIONES: COLLECTION

Las colecciones son grupos o conjuntos de datos o elementos sobre los que se definen operaciones de inserción, borrado y actualización. Collection es un interface que define las operaciones que debe tener una colección. No se puede reservar memora para un objeto de tipo Collection, ya que no es una case, de modo que son otras clases las que implementan el comportamiento/métodos de una colección. No se puede recorrer una colección a través de un índice, ya que no es un grupo ordenado de elementos.

Métodos de Collection más frecuentemente usados:

- boolean add(E ele), permite añadir el elemento ele a la colección
- boolean contains(Object obj), comprueba si el objeto obj se encuentra en la colección
- boolean isEmpty(), indica si la colección está vacía
- void remove(Object obj), elimina el objeto obj a la colección
- Iterator<E>iterator(), genera un iterador que se puede usar para recorrer la colección

Los métodos contains y remove hacen uso del método equals para comprobar si el objeto se encuentra en la lista.

Algunas implementaciones del interface Collection realmente no soportan todos sus métodos, generando una exception del tipo UnsoportedOperationException. Una de las formas de recorrer todos los elementos de una colección es a través de in bucle for-each. Las colecciones no se pueden modificar mientras se están recorriendo, en cuyo caso se genera una excepción del tipo ConcurrentModificationException.

COLECCIONES: ITERADORES

Un iterator es un interface en el que se definen un conjunto de operaciones que se pueden usar para recorrer los elementos de una colección. Para recorrer todos los elementos de se usa un puntero o iterador que señala al siguiente elemento de la colección, de modo que las operaciones básicas de un Iterador son:

- boolean hasNext(), indica si existe un elemento en la siguiente posición en la que se encuentra el puntero
- E next(), obtiene el elemento que se encuentra en la posición siguiente y actualiza el puntero a la posición siguiente
- remove(), elimina el elemento que está en la posición actual

Un iterador es una forma alternativa a for-each para recorrer los elementos de una colección. A diferencia de for-each, un iterador permite la eliminación de los elementos mientras se recorre la colección, eliminando tanto los elementos de la colección como del iterador. Un iterador solamente se puede usar una única vez para recorrer los elementos de la colección, ya que el puntero del iterador habrá llegado al final de la colección, ya que el puntero del iterador habrá llegado al final de la colección y hasNext devolverá siempre falso. El rendimiento de un iterador y un for-each es el mismo, ya que en realidad el compilador interpreta un for-each como si fuese un iterator.

4. Listas

Las listas son colecciones en las que los elementos tienen un orden que está relacionado con la secuencia en la que dichos elementos han ido siendo introducidos durante la ejecución del programa. Una lista mantiene siempre el orden en el cual se han introducido los elementos en la colección, de modo que a cada elemento se le asigna un índice que indica el lugar en el que ha sido introducido durante la ejecución del programa. List es un interface que, además de soportar las operaciones de una colección, define las operaciones que debe tener una lista.

Una lista se puede recorrer usando un índice para acceder a cada elemento de la colección ordenada. Una lista, al ser una colección, también se puede recorrer a través de un bucle tipo for-each. Una lista puede contener objetos duplicados, es decir, una referencia a un objeto puede aparecer en más de una ocasión en la colección. El interface List añade otros métodos a Collection, que están pensados para acceder a los datos a través de un índice

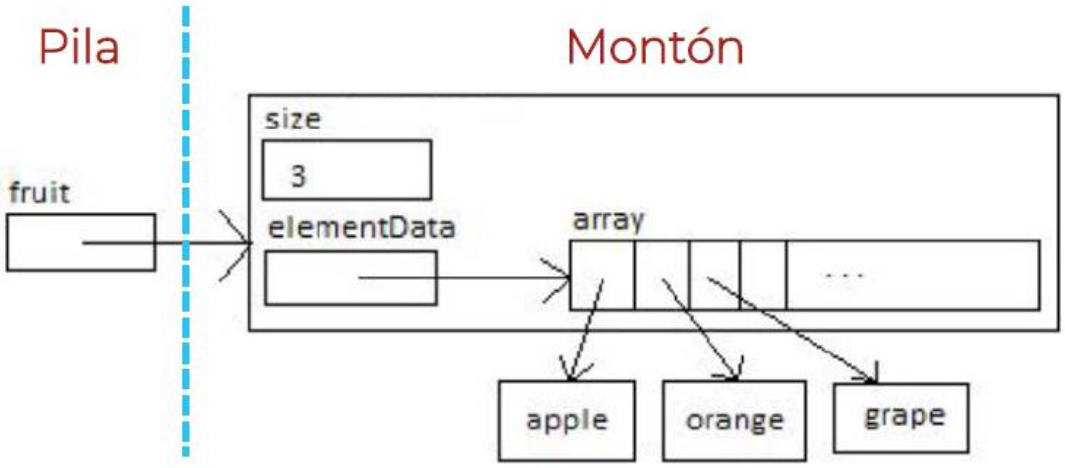
- E get(int i), obtiene el objeto que está en la posición i
- void set(int i, E ele), actualiza el objeto que está en la posición i
- E remove(int i), elimina el objeto que está en la posición i

LISTAS: ArrayList

La clase ArrayList es un tipo de lista que soporta la redimensión dinámica cuando el número de datos que se desea almacenar es mayor que la capacidad de almacenamiento de la lista. La redimensión es automática, de manera que no será necesario invocar ni programar ningún método para que esta redimensión tenga lugar. Si la redimensión se realiza con mucha frecuencia, se penalizará el rendimiento de la lista, ya que internamente esta redimensión supone incrementar el tamaño de un array de objetos ([]) a través de una operación de caché de datos (Arrays.copyOf). Un objeto de ArrayList puede contener objetos de tipo null.

El constructor sin argumentos de ArrayList reserva espacio para 10 elementos del tipo almacenado en la lista. Se diferencia entre tamaño del conjunto de datos que se desea almacenar y la capacidad de la lista. Cuando la lista alcanza el límite de su capacidad, por defecto aumenta dicha capacidad en 10 objetos. Si la lista se recorre a través del índice, es posible modificar su contenido, aunque se deberá tener en cuenta que un índice mayor que el tamaño de la lista no se asigna a ningún objeto. Al recorrer la lista es necesario comprobar que los objetos almacenados en ella son distintos de null.

La clase ArrayList encapsula un array de objetos al que se accede a través de los métodos definidos en el interface List. La referencia al array de objetos(fruit) se almacena en la pila, mientras que el array de objetos (elementData) y los otros atributos de la clase (size) se almacenan en el montón.



Las operaciones de acceso a los objetos del ArrayList tienen lugar en tiempo lineal con el numero de objetos almacenados en el array (O(n)). El aliasing tiene un mayor impacto al usar conjuntos de datos, ya que se pueden modificar sus elementos sin cambiar la dirección del conjunto.

5. Conjuntos

Los conjuntos son colecciones en las que no se repiten los datos, es decir, los datos son todos diferentes de acuerdo con el criterio de igualdad definido para los datos (método equals).

Set es un interface en el que se definen exactamente las mismas operaciones que en una Collection. Las implementaciones de Set deberán asegurar que para cada par de objetos de la colección se cumple la siguiente condición

```
Object e1, e2;
if(!e1.equals(e2)) return true;
```

En un Set puede haber, a lo sumo, un único null, aunque no todas las implementaciones del interface permiten almacenar null. Al ser una colección, para recorrer todos los elementos de un conjunto se hace uso de un iterador o de un bucle for-each. El criterio de igualdad es crítico en la gestión de los datos de los conjuntos, lo que en la práctica obliga a sobrescribir el método equals en las clases a las que pertenecen los objetos del conjunto. Se generan inconsistencias cuando estas modificaciones no se realizan con los métodos de Set.

CONJUNTOS: HashSet

La clase HashSet implementa el interface Set. Esta clase es un wrapper a la clase HashMap, de manera que internamente los datos se almacenan como las claves de un objeto HashMap. No

se asegura que los datos se guarden en el mismo orden en el que se insertan, si no que el orden de inserción está basado en su hash code.

Las operaciones de acceso a los objetos del HashSet tienen lugar en tiempo constante, de modo que no depende del número de objetos almacenados (O(1)).

6. Mapas

Un Map es un interface que relaciona una clave con un valor, de manera que a partir de la clave se obtiene el valor que está asociado a ella. Tanto las claves como los valores son objetos, es decir, no pueden ser tipos primitivos. Las claves no pueden estar repetidas. Los objetos sí que pueden estar repetidos, de modo que varias claves pueden tener asociado un mismo valor. Algunas implementaciones permiten almacenar claves y valores nulos.

La clave de un mapa puede ser cualquier tipo de objeto. Para localizar y obtener la clave con la que se recupera el valor se hará uso de equals, que deberá estar sobrescrito con un criterio de igualdad definido para la clase a la que pertenece la clave. La clave debería de ser un objeto inmutable, es decir, que no cambia una vez se realiza la reserva de memoria. Si el objeto no es inmutable, entonces es necesario controlar que no se modifiquen aquellos atributos usados en equals para comprobar la igualdad entre dos objetos.

Algunas implementaciones del interface Map realmente no soportan todos sus métodos, generando una exception del tipo UnsupportedOperationException.

Los métodos de Map más frecuentemente usados para acceder directamente a las claves y s los valores son

- V get(Object key), devuelve el valor asociado a una clave dada
- V put(K key, V value), almacena un valor al que se asocia una clave dada, quedando vinculado
- boolean containsValue(Object value), indica si un valor está almacenado en el mapa, es decir, si tiene asociada una clave
- boolean containsKey(Object key), indica si existe un dato que está asociado a una clave dada
- V remove(Object key), elimina la dupla <clave,valor> a partir de una clave dada.

Los objetos almacenados en un mapa no se pueden recorrer directamente, ya que el objetivo de los mapas es acceder a los datos a partir de las claves, no realizar una búsqueda de datos usando otros criterios diferentes de las claves.

El interface de Map define tres métodos que convierten las claves y los datos a colecciones (Collection) o a conjuntos (Set)

- Collection<V>values(), devuelve una colección que contiene los valores del mapa
- Set<K>keySet(), devuelve un conjunto con las claves del mapa
- Set<Map.Entry<K,V>entrySet(), devuelve un conjunto con todas las cuplas <clave, valor> del mapa.

Map.Entry es un interface que define métodos para acceder a una entrada de un mapa, es decir, a la dupla<clave,valor>

- K getKey(), permite acceder a la clave de la entrada en el mapa
- V getValue(), permite acceder al valor de la entrada en el mapa

MAPAS: HashMap

La clase HashMap implementa el interface Map. Los objetos que son las claves del mapa están asociados a un código hash (un entero) que se usa para comparar si dos claves son iguales, complementando así el método equals. Cuando se comparan dos objetos en un HashMap se aplica el siguiente procedimiento (lo mismo que para un HashSet); primero se comprueba si los objetos tienen el mismo hash code, de modo que si no es el mismo, se considerará que los objetos son diferentes. Si el hash code es igual, se invoca al método equals para aplicar el criterio de igualdad y comprobar que los objetos son iguales.

Se asume que aunque dos objetos no sean iguales, los hash code pueden ser iguales; pero si dos objetos son diferentes, entonces los hash code deben de ser diferentes. El uso de hash code simplifica y hace más eficiente el acceso a los datos en objetos HashMap (y HashSet), ya que en primer lugar se compara entre enteros y solo en el caso de que sean iguales tiene lugar una comparación más compleja.

Para generar el hash code, en Java es necesario sobrescribir el método hashCode(), que toda clase hereda de la clase Object. La implementación de hashCode() en la clase Object es un método nativo.

La clase HashMap implementa una tabla hash en la que los datos se almacenan de forma desordenada de acuerdo con una función hash. Las operaciones de acceso a los objetos del HashMap tienen lugar en tiempo constante, de modo que no depende del numero de objetos almacenados (O(1)).

7. Conjuntos de datos: Comparativa

Las características de los datos que se almacenan orientan la selección del tipo de conjunto de datos que se usarán en el programa.

Las listas ocupan menos memoria que los mapas y los mapas son más rápidos que las listas.

Collection	Ordering	Random Access	Key-Value	Duplicate Elements	Null Element
ArrayList	✅	✅	❌	✅	✅
LinkedList	✅	❌	❌	✅	✅
HashSet	❌	❌	❌	❌	✅
TreeSet	✅	❌	❌	❌	❌
HashMap	❌	✅	✅	❌	✅
TreeMap	✅	✅	✅	❌	❌

TEMA 4. Herencia: Jerarquías de clases, composición y abstracción

1. Concepto de herencia

La herencia es la segunda de las características principales de la Programación Orientada a Objetos que incorporan todos los lenguajes basados en este paradigma de programación.

Se puede definir la herencia como el mecanismo en virtud del cual una clase derivada reutiliza los atributos y los métodos que pertenecen a una clase base (o superior). Típicamente, la clase derivada tiene los mismos atributos y los métodos que la clase base, de manera que se "copian" implícitamente, es decir, se heredan, desde la clase base a la clase derivada. El objetivo de la herencia es la construcción de clases basadas en otras clases que ya tienen implementada la conducta deseada.

Los constructores de una clase no serán heredados por las clases derivadas, ya que se consideran que son específicos de la clase a la que pertenecen. Una clase derivada se considera una extensión de la clase base en la que sus atributos y métodos son los siguientes:

- Los métodos y atributos que se heredan de la clase base.
- Los métodos y atributos que se definen explícitamente como propios de la clase base.

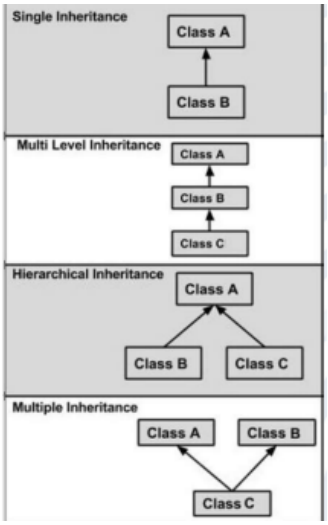
Entre una clase derivada (p.e, CartaDeMision) y una clase base (p.e, Carta), se dice que existe una relación del tipo "es-un/a"; por ejemplo, una CartaDeMision es una Carta.

Una clase **B** hereda todos los atributos y métodos de una clase **A**. Es el tipo de herencia más **básico**, en la que están basados los demás tipos de herencia

Una clase **C** hereda los atributos y métodos de una clase **B** que, a su vez, hereda de una clase **A**, de modo que **la clase C hereda los métodos y atributos de la clase A**

Las clases **B** y **C** heredan los atributos y los métodos de la clase **A**, de modo que **ambas clases se diferencian** en los atributos y métodos específicos de cada una de ellas

La clase **C** hereda los métodos y atributos de las clases **A** y **B**, de modo que es necesario definir **políticas de herencia** cuando A y B tienen métodos comunes



El principal beneficio de la herencia es la reutilización de código, en la que un trozo de código que ha sido previamente desarrollado, depurado y validado en una clase se usa en otra clase sin ser necesaria una modificación adicional. Adicionalmente, el uso de la herencia tiene otros beneficios:

- Simplifica el código, no siendo necesario implementar varias veces el mismo método en lo que serían las clases derivadas.

- Facilita el mantenimiento de los programas, ya que el cambio o reparación de los métodos tiene lugar en las clases base.
- Facilita la extensibilidad de los programas, ya que las nuevas clases(derivadas) se construyen a partir de otras clases (base).

La herencia también tiene algunas desventajas que pueden desaconsejar su uso generalizado en el desarrollo de las clases del programa. Si el nivel de jerarquización es muy profundo, el código es mucho más difícil de entender, tanto las relaciones entre las clases como los métodos que pertenecen a cada clase. Los cambios en las clases base tienen un claro impacto en las clases derivadas, de manera que esos cambios podrían ser inconsistentes con el código necesario para las clases base. Para facilitar la herencia es posible que los atributos deban tener un modificador de acceso diferente de privado que, por tanto, sería contrario al concepto de encapsulación.

2. Concepto de composición

La composición es el mecanismo mediante el cual una clase contiene objetos de otras clases a los que se les delega ciertas operaciones para conseguir una funcionalidad dada.

El principal beneficio de la composición es que reparte las responsabilidades entre los objetos, es decir, cada objeto se encarga de realizar unas operaciones dadas sobre los atributos, que después podrán invocar los demás objetos. Adicionalmente, la composición tiene otros beneficios; facilita el mantenimiento de los programas y la extensibilidad a través de la composición de clases. Otro beneficio importante es que en composición las clases están más desacopladas entre sí, de modo que un cambio en una de ellas tiene un impacto reducido en las otras.

El mecanismo de composición presenta también desventajas:

- Tiende a generar mucho más código y de mayor complejidad, para soportar la misma funcionalidad que se consigue cuando se aplica el mecanismo de herencia.
- Se necesita más tiempo de desarrollo en comparación con el mecanismo de herencia, puesto que la construcción de nuevas clases no está basada en otras clases, y por tanto, es necesario implementar los métodos en todas las clases.

3. Herencia vs Composición

Se dice que la composición entre dos clases es una relación de tipo “tiene-un/a”.

Podría parecer fácil determinar cuándo una relación entre dos clases es de herencia o de composición, pero en muchos casos no es tan sencillo. Desde un punto de vista de la programación, un empleado no debería de ser una persona, es decir, si un objeto empleado deja de ser empleado también dejará de ser persona, puesto a que, de acuerdo con la relación de herencia, un objeto empleado es una persona porque es un empleado; y por tanto, la identidad de un empleado no depende de la persona.

Es más consistente considerar que un empleado tiene los atributos de una persona, es decir, las funcionalidades de las dos clases no se “mezclan”, si no que la clase Empleado delega las operaciones relacionadas con las personas de la clase Persona.

4. Composición sobre herencia

¿Cuándo no se debería de usar herencia (y si composición)?

- Si las clases no están relacionadas desde un punto de vista lógico, la herencia no tiene sentido.
- Si una clase base tiene una sola clase derivada, la herencia no tiene sentido.
- Si las clases derivadas heredan código que no necesitan, la herencia no tiene sentido.
- Si existe la posibilidad de que las clases base cambien, la herencia complica el desarrollo del programa.
- Si es necesario sobrescribir muchos métodos de las clases base, la herencia complica el desarrollo del programa.

Caso	Diseño basado en herencia	Diseño basado en composición
Inicio del desarrollo	Más rápido	Más lento
Diseño del software	Más sencillo	Más complejo
Efectos no deseados	Ocurren con más frecuencia, sobre todo en cuando se trata de jerarquías profundas	Se reducen y están más localizados, en los métodos delegados
Adaptación a cambios	Para jerarquías profundas y con múltiples sobreescritura es más complicado	Más sencillo de cambiar, puesto que solamente afectan a las clases delegadas
Validación	Es difícil de realizar, sobre todo en jerarquías profundas y con múltiples sobreescrituras	Más sencillo de validar al estar el código muy localizado en las clases delegadas
Extensibilidad	Es sencillo, aunque se complica en jerarquías profundas	Tiene lugar de forma más sencilla a través de la composición de clases

5. Herencia en Java

En la herencia de Java se hace énfasis en que una clase derivada extiende los atributos y métodos de una clase base, aunque impone una serie de restricciones entre los dos tipos de clases. Es importante comprender que no existe una herencia múltiple, es decir, una clase derivada solo puede tener una única clase base de la cual extenderá sus atributos y métodos.

Solamente se heredan los atributos y métodos de la clase base que son públicos para la clase derivada, es decir, se heredarán en función de los tipos de acceso de los métodos y atributos de la clase base.

REVISANDO <tipo-acceso>:

Tipo de acceso	Comportamiento
private (privado)	La clase derivada **nunca heredará** los atributos y los métodos, independientemente del paquete en el que esté la clase base en relación a la clase derivada.
public (público)	La clase derivada **siempre heredará** los atributos y los métodos, independientemente del paquete en el que esté la clase base en relación a la clase derivada.
□ (acceso a paquete)	La clase derivada **solamente heredará** los atributos y los métodos de la clase base si ambas clases se encuentran en el mismo paquetes.
protected (protegido)	La clase derivada **siempre heredará** los atributos y los métodos, independientemente del paquete en el que esté la clase base en relación a la clase derivada.

Como los atributos privados no se heredan, para favorecer la herencia se podría pensar en que los atributos fuesen o bien públicos o protegidos. Aún con esa restricción los atributos deben ser privados, ya que se considera que la encapsulación es una característica que ofrece mayores beneficios que la herencia. Sin encapsulación el desarrollo de los programas se hace mucho más complejo y es mucho más difícil de mantener y validar; es decir, la composición no tiene sentido sin encapsulación.

Si los atributos son protegidos o públicos la encapsulación desaparece (acceso público) o se debilita (acceso protegido).

? Pregunta: si la clase derivada no hereda los atributos privados, pero sí hereda los métodos públicos de esa clase, ¿cómo es posible que la clase derivada tenga esos métodos públicos si no hereda los atributos que son usados por esos métodos?

⇒ Respuesta: en realidad en la clase derivada hereda todos los atributos, independientemente de su tipo de acceso, es decir, se heredan los atributos públicos, privados, protegidos y de acceso a paquete. El tipo de acceso especifica el nivel de visibilidad/accesibilidad que los métodos de la clase derivada tienen sobre los atributos y métodos de la clase base, es decir, si son privados, se heredan, pero no se puede acceder a ellos. El comportamiento es el habitual: los objetos reservan memoria e inicializan los atributos de la clase derivada a la que pertenecen, y los métodos que se heredan de la clase base acceden a dichos atributos para cada objeto.

? Pregunta: si la clase derivada no hereda los métodos privados de la clase base, pero sí hereda los métodos públicos, ¿cómo es posible que la clase derivada acceda a esos métodos públicos si no hereda los métodos privados que son usados por dichos métodos?

⇒ Respuesta: ocurre lo mismo que en el caso de los atributos, es decir, la clase derivada hereda todos los métodos de la clase base, independientemente del tipo de acceso que tengan. El tipo de acceso no evita o limita la herencia de los métodos, sino que impone restricciones sobre la visibilidad que tienen los métodos de la derivada sobre los métodos que se heredan de la clase base.

CONSTRUCTORES

Los constructores de la clase base no se heredan, puesto que se considera que contienen código específico de la clase a la que pertenecen ○ Los constructores tienen que reservar memoria e inicializar los atributos de la clase a la que pertenecen, pero el constructor de la clase base no tiene acceso a los atributos de la clase derivada, con lo cual si se hereda el constructor de la clase base, no podría ser invocado para crear objetos de la clase derivada

Si el constructor de la clase base no tiene argumentos ○ Cuando se invoca al constructor de la clase derivada, se invocará automáticamente el constructor de la clase base ○ Si se implementa un constructor sin argumentos para la clase derivada, obligatoriamente deberá de implementarse un constructor sin argumentos para la clase base, en caso contrario, el compilador de Java generará un error

Si el constructor de la clase base tiene argumentos; solamente se invocará a dicho constructor, puesto que no se puede garantizar que en la clase base exista un constructor con los mismos argumentos. ¿Cómo se logra la invocación de los constructores de la clase base que tienen argumentos?

Solución: el programador invoca manualmente al constructor de la clase base que considera oportuno en cada caso, lo que incluye al constructor que tenga los mismos argumentos que el constructor de la clase base desde la cual se invoca. Para ello, se hace uso de super.

super permite acceder desde la clase derivada a los atributos, métodos y constructores de la clase base sobre los que existe cierto nivel de visibilidad. super únicamente puede acceder a los elementos de la clase base que es inmediatamente superior a la clase derivada ○ Con super no se puede acceder directamente a los elementos de la clase que es la clase base (CartaDeEquipamiento) de la clase base (Infanteria) de la clase derivada (InfanteriaEuropa).

MÉTODOS Y SOBREESCRITURA

La sobreescritura de métodos es un mecanismo mediante el cual un método heredado de una clase base se implementa nuevamente en la clase derivada ○ La sobreescritura es necesaria cuando la implementación de un método de la clase base no es válida en la clase derivada o es conveniente adaptarla a las características específicas de la clase. Ejemplos de sobreescritura son los métodos toString, equals, hashCode y clone de la clase Object, que son implementados en las demás clases para adaptarlos a sus características. El nombre del método, sus argumentos y el tipo que devuelve deben ser los mismos en la clase base y en la derivada, mientras que el tipo de acceso debe ser el mismo o superior.

Todas las clases que se crean en un programa de Java son clases derivadas de la clase Object, lo que implica que heredan todos sus métodos.

- getClass indica la clase a la que pertenece el objeto que lo invoca.
- notify y wait están orientados a la gestión de los threads.
- finalize se invoca cuando el recolector de basura determina que no hay más referencias al objeto y lo elimina de la memoria del programa.

Una forma de sobrescribir los métodos en las clases derivadas es hacer uso de super para reutilizar la implementación del método que está en la clase base. Desde el método de la clase

derivada super invoca al mismo método de la clase base, es decir, al método que se sobrescribe y que tiene el mismo nombre y argumentos ○ Se usa super porque la clase base no es capaz de distinguir entre la implementación de la clase base y la implementación de la clase derivada, con lo cual la única forma de reutilizarlo es a través de super. El método de la clase base se hereda, pero al sobrescribirse no se puede acceder a su implementación en la clase base.

6. Clases Abstractas

Las clases abstractas son clases que no se pueden instanciar ○ Pueden tener constructores, pero no se pueden invocar a través de new, de modo que únicamente se pueden invocar mediante super cuando son las clases base de otras clases.

Ejemplo: si CartaDeEquipamiento( ) es un constructor de una clase abstracta, entonces new CartaDeEquipamiento( ) no está permitido

Pueden tener atributos y métodos implementados de la misma forma en la que los tienen clases que no son abstractas. Habitualmente, se usan como clases base de otras clases de las que sí se pueden crear objetos, ocupando los primeros niveles de las jerarquías de clases y facilitando con ello la reutilización de sus atributos, métodos y constructores en las clases derivadas.

Las clases abstractas deben tener implementados la mayor cantidad posible de métodos para que las clases derivadas puedan heredarlos y así favorecer la reutilización de código. Una clase debería ser abstracta cuando todos los objetos pertenecen a cualquiera de las otras clases de la jerarquía, no siendo necesario crear un objeto de la clase abstracta.

Las clases abstractas pueden tener métodos abstractos. Son métodos que no tienen cuerpo, es decir, métodos que no están implementados y en los que solamente se especifica su nombre, lo que devuelve y los argumentos de entrada “public abstract nombre_método(nombre *) ;”

Si una clase tiene al menos un método abstracto entonces debe ser necesariamente una clase abstracta. Los métodos abstractos deben ser implementados en las clases derivadas de la clase abstracta a la que pertenecen, siempre que dichas clases no sean abstractas.

Los métodos abstractos no tienen cuerpo, pero pueden ser “invocados” desde los constructores de las clases abstractas. Los constructores de las clases abstractas solamente pueden ser invocados desde constructores de las clases derivadas de esas clases abstractas. Si los constructores no van a ser invocados de forma directa, es posible “requerir la invocación” de métodos abstractos, es decir, indicar que se tiene que ejecutar una implementación de ellos. El método abstracto deberá estar implementado necesariamente en las clases derivadas de la clase abstracta, de modo que será esa implementación la que se invocará en el constructor de la clase abstracta.

Una clase abstracta…

- Puede tener todos los métodos abstractos. Debería tener tantos métodos implementados como sea posible
- Puede tener todos los métodos implementados.
- Puede ocupar cualquier lugar de la jerarquía de clases. Debería ocupar los niveles superiores de la jerarquía de clases.

- Puede ser una clase derivada de una clase no abstracta. Su clase base debería ser una clase abstracta.
- Puede no tener constructores. Debería tener tantos constructores como sea necesario.
- Puede tener atributos de cualquier tipo

Puede tener todos los métodos abstractos

- En este caso el énfasis en el diseño del programa se hace en que todas las clases derivadas de la clase abstracta tengan todas los mismos métodos que la clase abstracta, es decir, que tengan una interfaz común.
- Facilita el mantenimiento y extensibilidad del programa a través de la independencia entre las clases.

Debería tener tantos métodos implementados como sea posible

- En este caso el énfasis en el diseño del programa se pone en la reutilización de código donde todas las clases comparten la misma implementación de los métodos que la clase base, reduciendo los tiempos de desarrollo.

Típicamente, una jerarquía de clases se construye con clases abstractas (primeros niveles), con clases instanciables (niveles intermedios y finales) y con clases finales. Las clases finales son clases de las cuales no se pueden crear clases derivadas, es decir, son las últimas clases o nodos de la jerarquía de clases. Una clase solamente debería ser final cuando se garantiza que no será necesario crear clases derivadas de ella.

La palabra final en Java indica que el elemento que cualifica no se puede modificar.

- Si el elemento es un método, entonces si el método pertenece a una clase base, no podrá ser sobrescrito en una clase derivada, es decir, una vez tiene una implementación, no se podrá modificar.
- Si el elemento es un atributo, entonces una vez el atributo haya tomado un determinado valor, no podrá ser modificado nuevamente Los atributos finales no pueden tener setters porque solo pueden modificar su valor una única vez.

Los atributos finales se usan para implementar las constantes del programa, puesto que, al establecer un valor para ellos, ya no se podrán volver a modificar. Típicamente, una constante se define con final static, ya que así es posible hacer uso de la constante sin que sea necesario crear un objeto de la clase en la que está definida. La palabra static usada como modificador en atributos o métodos hace que se almacenen en la memoria estática, permitiendo que estén disponibles desde el inicio del programa sin necesidad de tener que crear objetos, es decir, de instanciar las clases.

TEMA 5. Polimorfismo: Jerarquías de Clases y Abstracción

1. Polimorfismo en herencia

El polimorfismo en herencia es el mecanismo mediante el cual un objeto se puede comportar de múltiples formas en cada momento, en función del contexto en el que parece dicho objeto dentro del programa.

- Un objeto se puede comportar como una instancia de la clase a la que pertenece, es decir, de la clase cuyo constructor se invoca a través de new.
- Un objeto se puede comportar como una instancia de alguna de las clases que se encuentran en un nivel superior de la jerarquía en relación a la clase a la cual pertenece realmente el objeto.

El polimorfismo está relacionado con la herencia y con el uso de los métodos de las clases que se heredan/sobrescriben.

¿Existe una contradicción en el hecho de que una clase sea abstracta y que un objeto se comporte como dicha clase?

La restricción que se impone sobre una clase abstracta es que no se pueden crear objetos de dicha clase; en otras palabras, no se puede usar new para invocar a los constructores de las clases abstracta Las clases abstractas no solamente se usan para estructurar las jerarquías de clases, sino que también para restringir los métodos que pueden invocar los objetos.

2. Upcasting & Downcasting

Cuando se indica que un objeto se comporta como una clase de la jerarquía diferente a la que pertenece, se está forzando un cambio en el tipo de dato del objeto.

- En upcasting un objeto de una clase derivada se comporta como una de las clases base de la jerarquía.
- En downcasting un objeto de una clase base se comporta como una de las clases derivadas de la jerarquía.

Este cambio de tipo de dato no implica una nueva reserva de memoria para el objeto, sino que el objeto seguirá siendo el mismo, independientemente de los cambios de tipo de dato que tengan lugar a lo largo del programa. El casting de objetos no solamente conlleva un cambio en el tipo de dato, sino que también afecta a la accesibilidad de los atributos y de los métodos que se pueden invocar desde el objeto.

Los únicos métodos que serán accesibles por el objeto son (1) los que están definidos en la clase a la cual se ha realizado el cast; y (2) los que hereda dicha clase de las clases base de la jerarquía.

3. Upcasting

Upcasting es el mecanismo más habitual con el que se facilita el polimorfismo, ya que parece natural pensar que un objeto de la clase derivada se comporta como una de su clase base. Cuando se realiza un upcasting no es necesario indicar de forma explícita un cast entre la clase derivada y la clase base.

Los métodos y atributos que son propios de la clase a la que pertenece el objeto no son accesibles, es decir, se pierde la visibilidad de esos métodos y atributos, puesto que el objeto deja de comportarse como esa clase.

Si un método está sobrescrito cuando se realiza upcasting, el objeto usará la implementación del método correspondiente a la clase a la que pertenece el objeto, de modo que, aunque se comporta como otra clase, no utiliza las implementaciones de los métodos de esa otra clase. Si se usase la implementación de los métodos correspondientes a la clase sobre la que se realiza upcasting, no existiría polimorfismo cuando se tratase con clases abstractas.

Upcasting es un concepto clave en programación orientada a objetos, puesto que facilita la toma de decisiones con relación al diseño del programa. Si en un programa es necesario el uso de upcasting, entonces se deberá de aplicar el mecanismo de herencia.

El mecanismo de upcasting nunca producirá un error, ya que con la herencia se garantiza que las clases derivadas tienen los mismos métodos que la clase base.

- Una vez se ha realizado upcasting sobre un objeto, no hay ningún modo de acceder a los métodos específicos de la clase a la cual pertenece dicho objeto.
- Para aprovechar todo el potencial del upcasting el diseño de los programas debe ser muy cuidadoso, eligiendo qué métodos se deben sobrescribir y cuáles son específicos de las clases.
- Las clases abstractas permiten definir qué métodos es necesario que tengan las clases que derivan de ellas, facilitando con ello el uso del upcasting para la simplificación de los programas

4. Downcasting

Cuando se realiza downcasting es necesario indicar de forma explícita a qué clase derivada se "convertirá" el objeto de la clase base, es decir, en el código deberá de especificar el tipo de datos del objeto. Se usa un cast con el que se fuerza la conversión de los tipos de datos desde la clase base a la derivada, de modo que ese cast se especificará con el siguiente formato: (<nombre_clase_derivada>). El compilador no comprueba en tiempo de diseño si el cast tendrá lugar de forma correcta, ni tan siquiera comprueba que el objeto se encuentre en la misma jerarquía que la clase derivada a la que se "convierte".

El principal problema de la operación de downcasting es que no se puede garantizar que el objeto disponga de los mismos métodos y atributos que los de la clase derivada a la cual se realiza el casting.

- La operación de casting tiene lugar en tiempo de ejecución y en ella solo se comprueba si el tipo del objeto es la clase derivada.
- Aunque el objeto soporte los mismos métodos y atributos que los de la clase derivada, si el tipo del objeto no es la clase derivada, la operación de downcasting generará una excepción.
- Se debe comprobar en tiempo de ejecución que el tipo de dato del objeto sea la propia clase derivada a la que se realiza el casting (operador instanceof).

El operador instanceof determina en tiempo de ejecución si el tipo de dato de un objeto es una clase dada, de manera que si devuelve cierto se podrán invocar los métodos de dicha clase.

5. Beneficios del polimorfismo

Facilita la simplificación del código del programa, puesto que los objetos se pueden manejar como objetos de las clases base, de modo que la invocación de los métodos comunes a todas las clases se realiza desde una misma clase (la clase base).

Facilita la extensibilidad de los programas, ya que la inclusión de nuevas clases en la jerarquía que implementan los métodos comunes de las clases base no supondrá modificaciones en el código del programa. Con downcasting la introducción de nuevas clases podría implicar cambios en el código del programa, pero serían extensiones a los métodos en los que ya se manejan las distintas clases derivadas de las clases base.

TEMA 6. Interfaces: Abstracción e independencia de clases

1. Concepto de Interfaz

Los interfaces son uno de los componentes clave en el diseño de los programas, ya que con ellos se establecen de forma muy precisa los requisitos que deben cumplir las clases del programa.

- Se definen los tipos de datos que se deben crear en el programa, de modo que pueden existir otros tipos de datos, pero al menos deben estar definidos los indicados en los interfaces.
- Se definen exactamente los métodos que deben tener las clases del programa, indicando exactamente sus nombres, los tipos de argumentos que reciben y el tipo de datos que devuelven.
- Cada clase puede tener más métodos públicos y privados, pero los métodos de interés son los indicados en los interfaces.

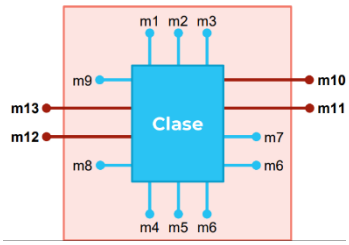
Un interfaz se entiende como un compromiso o acuerdo entre programadores aplicación para que las clases que crean unos se puedan usar sin errores ni mayores adaptaciones por otros

⇒ Programador 1: "En el interfaz se indica exactamente lo que necesito que hagan las clases que tienes que programar. No me interesa saber cómo vas a hacerlo, mientras lo hagas de la forma en la que se indica en el interfaz y que la implementación sea eficiente"

? Programador 2: "¿Puedo crear otras clases de apoyo al interfaz que tengo que implementar? ¿Puedo crear otros métodos además de los que se indican en el interfaz?

⇒ Programador 1: "Puedes hacer lo que creas conveniente, mientras cumplas con el compromiso al que hemos llegado".

El objetivo de un interfaz es establecer los métodos que una clase debe implementar y que serán los métodos visibles y accesibles por las otras clases del programa. No solo se trata de los métodos públicos de las clases (m1 al m13). Se trata de métodos públicos que proveen la funcionalidad requerida por las otras clases del programa (m10 al m13). Los otros métodos no son de interés para las otras clases del programa (m1 al m9).



El uso de interfaces facilita enormemente el desarrollo y el mantenimiento de los programas, puesto que con ellos se dividen las responsabilidades entre los programadores y permiten una implementación independiente de las clases.

- Un cambio en una clase en la que se implementan los métodos indicados en un interfaz no afectará a las otras clases que hace uso de dicho interfaz.
- Un cambio en el interfaz afectará de forma significativa tanto a las clases que implementan el interfaz como a las clases que lo usan. Los interfaces deberían ser invariantes durante el desarrollo de un programa; una vez se definen, no deberían cambiar.

2. Interfaces en Java

Un interfaz se entiende como una plantilla de una clase que contiene constantes, declaraciones de métodos abstractos, métodos por defecto y métodos estáticos.

- No se puede crear objetos de interfaces, con lo cual no tienen constructores ○ Las clases deben implementar los métodos abstractos del interfaz para que el interfaz pueda ser usado en otras clases.
- Una vez el interfaz haya sido implementado, se comportará como una clase, pudiendo invocar a los métodos abstractos, con la implementación de clase, los métodos estáticos y por defecto.

Si una clase implementa un interfaz, entonces esa clase tendrá los mismos métodos que el interfaz, del mismo modo que una clase derivada tiene los mismos métodos que una clase base.

- La clase que implementa el interfaz se puede entender que es como una clase derivada del interfaz, puesto que un interfaz es equivalente a una clase abstracta con las siguientes características.
    - Tiene todos sus métodos abstractos.
    - Tiene todos sus atributos constantes.
    - No tiene constructores implementados.
- Se pueden aplicar todos los conceptos de herencia, jerarquías de clases, clases abstractas y polimorfismo.

3. Clases abstractas vs Interfaces

Clases abstractas	Interfaces
Se definen cuando las clases derivadas tienen algunos métodos comunes	Se definen cuando las clases que los implementan tienen diferentes implementaciones para diferentes objetos
La herencia múltiple no es posible	Se puede entender que la herencia múltiple es posible, aunque solo puede haber una clase base
Pueden tener métodos abstractos y métodos concretos	Solamente tienen métodos abstractos, además de métodos estáticos y métodos por defecto
Los métodos abstractos pueden ser públicos y protegidos	Los métodos abstractos deben de ser públicos
Tienen constructores	No tienen constructores
Pueden tener cualquier tipo de atributo	Los únicos atributos que puede tener son de tipo estático y final (son constantes)

4. Métodos por defecto

El principal problema de los interfaces es la incapacidad de las clases que los implementan para adaptarse a los cambios que tengan lugar en ellos.

⇒ Al añadir nuevos métodos abstractos y/o actualizar los métodos que están declarados en un interfaz, las clases que implementan dicho interfaz generarán errores de compilación.

Existirán casos en el programa en las cuales no se usarán los nuevos métodos del interfaz, mientras que en otras sí que será necesario utilizarlos.

Los métodos por defecto resuelven parcialmente el problema de adaptación de las clases a los cambios en los interfaces que implementan.

Los métodos por defecto son métodos que se deben declarar e implementar en el interfaz, de modo que se heredarán por las clases que implementan dicho interfaz.

- La implementación debe operar únicamente con los argumentos del método por defecto, ya que en el interfaz no existen atributos que no sean constantes.
- Los métodos por defecto pueden ser sobrescritos en las clases que implementan el interfaz o en las clases derivadas de ellas.

Se debe de valorar la conveniencia de definir métodos por defecto, sobre todo si necesitan argumentos asociados a atributos de las clases que implementan los interfaces que contienen dichos métodos. El código se hace más complejo de entender y mantener, puesto que los métodos de las clases tienen argumentos que no serían necesarios al hacer referencia a los atributos de dichas clases.

Los métodos por defecto serán heredados por las clases que implementan al interfaz que los contiene y por los interfaces derivados de dicho interfaz. Los métodos por defecto, al ser heredados, también podrán ser sobrescritos por las clases y los interfaces derivados. En la sobreescritura se puede hacer uso de super siguiendo el siguiente formato: <nombre_interfaz>.super.<método>.

5. Métodos estáticos

Los métodos estáticos son métodos que pueden ser invocados desde el inicio del programa, de modo que para invocarlos no es necesario crear ningún objeto de la clase en la que se han definido.

- **Se pueden definir en cualquier clase**, sea abstracta, instanciable o final, o en cualquier interfaz.
- Se cargan automáticamente en memoria al arrancar el programa.
- En un método estático únicamente se pueden invocar métodos de la clase o del interfaz que sean estáticos, puesto que tienen que estar disponibles desde el arranque del programa.
- No son heredados por las clases e interfaces derivados de ellos.

6. Métodos estáticos vs Métodos por defecto

Métodos por defecto	Métodos estáticos
Solamente pueden estar definidos en los interfaces	Se pueden definir en interfaces y en clases
Pueden invocar cualquier tipo de método, incluyendo métodos por defecto, estáticos y abstractos	Solamente pueden invocar métodos que, a su vez, sean métodos estáticos
Son heredados por las clases y por los interfaces derivados del interfaz que los contiene	No pueden ser heredados, siendo métodos que son específicos de las clases o de los interfaces en los que están implementados
No están disponibles en el arranque del programa, es necesario crear un objeto para invocarlos	Están disponibles al arrancar el programa

7. Herencia e interfaces

Si una clase abstracta cumple las mismas condiciones que un interfaz "clásico", es decir, solamente tiene métodos abstractos y no tiene ningún atributo que no sea constante, entonces se podría pensar que una clase derivada tiene varias clases base, o lo que es lo mismo, que existe herencia múltiple.

Una clase debe de implementar todos los métodos abstractos de todos los interfaces que implementa y hereda todos los métodos por defecto de todos esos interfaces.

- Si varios interfaces tienen en común algún método abstracto, la implementación de ese método en la clase será válida para todos los interfaces.
- Si varios interfaces tienen en común algún método estático, no existirá ningún conflicto en la clase, ya que ese tipo de métodos no serán heredan.
- Si varios interfaces tienen en común algún método por defecto, existirá una colisión entre dichos métodos con relación a cuál de ellos será el que heredará la clase.

¿Cómo se resuelve la herencia de los métodos por defecto cuando existen varios interfaces base, es decir, cuando una clase implementa varios interfaces?

Para resolver el conflicto, la clase que implementa los interfaces deberá de sobrescribir los métodos por defecto que son comunes a dichos interfaces.

- No existe ningún mecanismo automático para seleccionar cuál es la implementación de los métodos por defecto que heredan las clases.
- Realmente en la propia sobreescritura de los métodos por defecto se puede realizar la selección de la implementación de los métodos por defecto, utilizando para ello super.

Se pueden crear jerarquías de interfaces siguiendo las mismas reglas de herencia que en el caso de las jerarquías de clases. Un interfaz derivado puede tener varios interfaces base, es decir, existe herencia múltiple entre interfaces. Puesto que en un interfaz tiene todos sus métodos implícitamente públicos, los interfaces derivados heredarán todos los métodos de los interfaces base, excepto los métodos estáticos. Existe sobreescritura de los métodos por defecto, de

manera que los interfaces derivados pueden usar super para invocar la ejecución de los métodos por defecto de los interfaces base. No se puede establecer herencia entre clases e interfaces, es decir, una clase no puede extender un interfaz ni viceversa.

Si varios interfaces base tienen en común algún método por defecto, existirá una colisión entre dichos métodos con relación a cuál de ellos será el que heredará el interfaz derivado. La forma de resolver este conflicto es la misma que en el caso de las clases que implementan varios interfaces que tienen algún método por defecto en común.

Las jerarquías combinan

- La **herencia de clases**
- La **herencia de interfaces**
- La **implementación de interfaces por parte de las clases**

Las jerarquías resultantes pueden ser muy complejas pero lo importante es que sean extensibles de forma relativamente sencilla.

TEMA 7. Excepciones: Gestión de errores y excepciones

1. Concepto de excepción

Una de las tareas importantes que es necesario realizar en el desarrollo de un programa es la gestión de los errores que ocurren durante su ejecución, ya que su correcto tratamiento facilita la usabilidad y mantenimiento del programa. En la programación de los métodos se debe de favorecer la separación lógica de los diferentes tipos de código.

- Código asociado **a la funcionalidad provista por el método**.
- Código asociado **a la interacción con usuarios u otros programas**.
- Código asociado **al almacenamiento y acceso a los datos**.
- Código asociado **a la gestión de errores**.

Una excepción es la ocurrencia de errores y/o situaciones de interés durante la ejecución de los métodos que proporcionan la funcionalidad del programa. Las excepciones no solo están relacionadas con lo que se entiende habitualmente por errores, sino también con situaciones a las que se debe dar respuesta y que forman parte de la lógica del programa.

- Una división en la que el cociente es cero es un error.
- La introducción de un comando incompleto no es un error, sino una condición que puede tener lugar en la ejecución del programa y que es necesario tratar.

El mecanismo de Java para el tratamiento de las excepciones aplica la filosofía de separar el código para la detección, la comunicación y el tratamiento de las excepciones del código de los métodos que proporciona la funcionalidad del programa.

2. Definición de excepciones

Antes de especificar las etapas correspondientes a la gestión de las excepciones en tiempo de ejecución es necesario (1) definir qué excepciones pueden tener lugar; y (2) definir en qué métodos ocurrirán dichas excepciones. Una excepción se especifica como una clase derivada de la clase Throwable, cuyos métodos más usados habitualmente son los siguientes:

- String getMessage( ) devuelve el mensaje que se ha indicado en la ocurrencia de la excepción.
- void printStackTrace( ) imprime en consola la secuencia de invocaciones de métodos que ha llevado a la ocurrencia de la excepción.

En Java existen dos tipos de excepciones; chequeadas y no chequeadas:

- **Excepciones chequeadas** (“checked”), se comprueba en tiempo de compilación en qué métodos puede ocurrir una excepción, en cuyo caso esos métodos deben indicar de forma explícita esa posible ocurrencia. Son clases derivadas de la clase Exception.
- **Excepciones no chequeadas** (“uncheked”), no se comprueba en tiempo de compilación cuáles son los métodos en los que tiene lugar la ocurrencia de la excepción, descargando en el programador la responsabilidad de chequear dónde puede ocurrir para intentar capturarla y tratarla. Son clases derivadas de las clases RuntimeException o Error.

Si el error o la condición que ha provocado la ocurrencia de la excepción no impide la ejecución del programa, la excepción debería ser del tipo chequeada. Ejemplo: NullPointerException es no chequeada porque en pocas ocasiones se puede" arreglar" un nulo de un objeto.

La definición de excepciones como clases derivadas de la clase Exception confiere a los programadores un mayor control sobre la gestión de dichas excepciones, puesto que obliga a que los métodos declaren qué excepciones podrían ocurrir durante su invocación.

- public Exception(String message). message es el mensaje con el que típicamente el programador explica por qué ha tenido lugar la ocurrencia de la excepción, que es el momento en el que se crea el objeto de tipo clase derivada de Exception. Este mensaje es lo que habitualmente se le muestra al usuario.
- Este constructor debería ser invocado desde el constructor de la clase derivada de Exception.

Para las excepciones chequeadas, el compilador exige que se indiquen explícitamente los métodos en los que tienen lugar dichas excepciones.

- Se deben de etiquetar los métodos en los que tienen lugar las excepciones.
- Un mismo método puede estar etiquetado con más de un tipo de excepción, es decir, con más de una clase derivada de Exception.
- Un constructor también puede generar excepciones.

En la signatura del método se deberán de indicar todas las excepciones que podrían ocurrir durante la ejecución del método, es decir, todas las excepciones que se generarán. Si las excepciones pertenecen a la misma jerarquía, es decir, tienen una misma clase base, sería suficiente con etiquetar el método con la clase base o clases superiores a ella.

3. Gestión de excepciones: Intentar (try)

La gestión de las excepciones comienza con la invocación de un métodoA desde otro métodoB, de modo que se puede decir que el métodoA intenta ejecutar con éxito el método. Es necesario indicar en qué parte del código del método A se va intentar la ejecución con éxito del métodoB, es decir, se intenta la invocación del métodoB esperando obtener el resultado por el cual tiene lugar dicha invocación.

try{

En el bloque try se deberá de invocar obligatoriamente al métodoB, aunque lo más deseable es que todo el código funcional del método A se sitúe dentro de un único bloque try para que sea más legible}

En el código de un puede haber tantos bloques try como invocaciones a métodos que generan una excepción, incluso aunque sean varias invocaciones a un mismo método, aunque con un único bloque try se hace más legible el código del método.

4. Gestión de excepciones: Lanzar (throw)

Una vez se invoca la ejecución del métodoB, el control del programa pasa al código de dicho método, que tiene dos partes: (1) la detección y (2) la generación de la excepción para que sea tratada en el método que realizó la invocación.

La detección de la excepción consiste en especificar dentro del código del métodoB las condiciones en las cuales se genera la excepción.

- Chequear si el numerador de la división es 0.
- Chequear los códigos de error descritos en el proyecto del Risk, que, en realidad, no son errores, sino excepciones que ocurren durante la ejecución de los métodos.

Al generar la excepción se completa lo que se entiende por lanzar la excepción, que consiste en crear un objeto del tipo de excepción que se transferirá al trozo de código en el cual tiene lugar el tratamiento de la excepción.

- Cuando se genera la excepción, es decir, cuando se crea el objeto del tipo excepción, es habitual incluir en ella información sobre el contexto en el que ha ocurrido, incluyendo (a) una descripción textual sobre la causa que la ha provocado; y/o (b) referencias a objetos que se podrán usar en el tratamiento de la excepción.
- Al lanzar una excepción se interrumpe la ejecución del método en el mismo punto en el cual se ha lanzado, es decir, el flujo de programa abandona el método y apunta directamente al código en el que se lleva a cabo el tratamiento de la excepción.

La palabra throw se utiliza para indicar que se interrumpe la ejecución del método y se lanza la excepción para que pueda ser capturada por el código en el que se trata la excepción, que es el punto en el que continúa la ejecución del programa. La información del contexto de ocurrencia de la excepción se pasa en los argumentos del constructor de la clase asociada a la excepción.

5. Gestión de excepciones: Tratar (catch)

Una vez se lanza una excepción, el objeto de tipo excepción se transfiere como “argumento” al código en el que se realiza su tratamiento, de modo que dicho código tiene información contextual sobre la ocurrencia de la excepción.

El momento en el que el flujo de programa entra en el código para el tratamiento de la excepción se denomina captura de la excepción. Típicamente, el tratamiento consistirá en mostrar al usuario y/o grabar en un fichero el mensaje de texto que se ha pasado como argumento al constructor en la creación del objeto que se lanza cuando ocurre la condición de la excepción.

El tratamiento de la excepción tiene lugar en los denominados bloques catch, que están directamente asociados a los bloques try, de modo que un boque catch no puede existir sin haber especificado previamente un bloque try. En el bloque catch hay código que realiza el tratamiento de la excepción, usando la información de contexto que contiene el objeto <nombre_objeto> de la clase <tipo_excepcion>.

RESTRICCIONES ENTRE LOS BLOQUES try–catch

- Si existe un bloque catch deberá existir necesariamente un único bloque try.
- Un bloque try deberá tener asociado, al menos, un bloque catch.
- Un bloque try podría tener tantos bloques catch como el número de tipos de excepciones se puedan lanzar en el bloque try.
- Un bloque try podría tener menos bloques catch que el número de excepciones que se hayan lanzado en el bloque try.
    - Si las excepciones que se lanzan tienen la misma clase base, entonces puede definirse un único bloque catch.
    - Si el tratamiento es el mismo puede usarse multi-catch.

La opción multi-catch no añaden ningún tipo de semántica a la gestión de la excepción, sino que está orientada a simplificar el código del bloque catch.

La herencia y el polimorfismo juegan un papel importante a la hora de decidir qué bloque catch se ejecutará cuando se lanza una excepción. Si el argumento del bloque catch es una de las clases superiores a una claseA, después de ese catch no podrá existir un catch con la claseA, puesto que la excepción se habrá capturado en el catch de la clase superior.

En el tratamiento de las excepciones, además de los bloques catch también existe el bloque finally, que se ejecuta siempre; independientemente de la excepción que se haya capturado o incluso aunque no se haya capturado ninguna excepción.

- Un bloque try puede tener asociado un único bloque finally.
- El bloque finally siempre deberá ir después de todos los bloques catch asociados al tratamiento de la excepción.
- No es posible definir un bloque finally si no existe, al menos, un bloque catch el que se capturen las excepciones generadas en el bloque try.
- Uno de los usos habituales del bloque finally es liberar recursos.

6. Revisitando generación de excepciones

Un métodoA que invoca a un métodoB que lanza una excepción no debe tener necesariamente un try–catch como parte de su código, es decir, el métodoA no tiene por qué realizar la captura de la excepción. El métodoA puede delegar el tratamiento de la excepción a otro métodoC que lo invoque, de modo que existe un encadenamiento lanzamientos de excepciones que tiene como fin la unificación del código en el cual se realizar el tratamiento final de la excepción.