3 Programación Orientada a Objetos

Comencemos este camino interminable de aprendizaje acerca de la programación orientada a objetos (POO) con los conceptos básicos de este tipo de programación -trataremos que sin pérdida de generalidad mostraremos los ejemplos en el lenguaje Java-, para ello iniciemos con lo que entendemos por un objeto.

¿Qué es un objeto? La definición más básica dice: "Es un ente computacional que puede contener datos y comportamientos".

> Ejemplo: un polinomio, una matriz o un número complejo. Cada uno de estos objetos tiene un comportamiento propio.
>
> Ejemplo: un polinomio puede sumar/integrar/derivar/evaluar.

Por lo tanto, un objeto lo podemos definir formalmente como: ente computacional que puede contener datos y exhibe comportamientos.

Mensaje Todos los objetos de los cuales hablamos tienen comportamientos y se relacionan con otros objetos (se piden cosas entre ellos). Por lo cual podemos definir a un mensaje como: Interacción entre dos objetos: un emisor E y un receptor R. Un emisor le envía un mensaje a un receptor. El emisor puede obtener o no una respuesta.

> Ejemplo: evaluar/grado/integrar.

Ciclo de vida de un objeto Todos los objetos tienen una vida, se puede decir que nacen cuando son instanciados y mueren cuando se elimina de la memoria. No obstante, existen dos formas de eliminar de memoria a un objeto dependiendo del lenguaje de programación. Los mecanismos son: Recolector de Basura (*Garbage Collector)* y *Destructores*.

Recolector de Basura: Es un mecanismo que se encarga de borrar de la memoria las referencias a objetos y entidades que ya no se usan más, de manera que se pueda maximizar el uso del espacio en memoria.

Destructores: Son métodos que se definen para cada objeto y cuyo principal objetivo es liberar los recursos que fueron adquiridos por el objeto a lo largo de su ciclo de vida y romper vínculos con otras entidades que puedan tener referencias a él.

Métodos ¿Que es un método? ¿En qué difiere de un mensaje?

Un método es la sección de código que se ejecuta al enviar un mensaje. Se identifica con una firma -el nombre y los parámetros del método-, que es la misma firma del mensaje enviado. Entonces, cuando un objeto recibe un mensaje, se ejecuta un método cuya firma es la misma que la del mensaje.

Método de la clase Polinomio:

```
// Evalua el polinomio en el valor pasado como parametro
public double evalua(double x) {
   int i;
   double p = 1.0, r = 0.0;
   for (i = 0; i < Dim; i++) {
      r += coeficiente(i) * p;
      p *= x;
   }
   return r;
}
```

La firma de un objeto se define con tres componentes:

1. El nombre del método.
2. Los parámetros que recibe el método.
3. Lo que devuelve el método (que puede ser nada u otro objeto).

Clases Es común que empezemos con la definición de "Clase" en los textos. Esto se debe a que las clases solo son una forma de implementar objetos, pero no son la única manera de hacerlo (como veremos más adelante), por lo que es fundamental que no pensemos automáticamente en clases cuando hablamos de objetos. Sin embargo, la mayor parte de los lenguajes de programación que usamos -laboral y académicamente- usan clases.

¿Qué es una clase? Podemos definir a una clase utilizando dos definiciones complementarias: una clase es un molde, a partir del cual se crean los objetos. Cuando instanciamos un objeto, el ambiente le pregunta a dicha clase que características y métodos debe tener el objeto. La otra definición es: una clase es un ente que determina el comportamiento y el tipo al que pertenecen sus instancias.

Clase Persona:

```
public class Persona {
  private String nombre;
  private String apellidos;
  private int edad;
  //Constructor
  public Persona(String nombre, String apellidos, int edad) {
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }
  public String getNombre () {
    return nombre;
  }
  public String getApellidos () {
    return apellidos;
  }
  public int getEdad () {
    return edad;
   }
  public void visualiza() {
    System.out.println ("Nombre: " + getNombre() + " " +
     getApellidos() + " edad: " + getEdad());
  }
  public static void main (String [ ] Args) {
    Persona p = new Persona ("Mauro", "Ruiz perez", 36);
    p.visualiza();
  }
}
```

Características de la Programación Orientada a Objetos Los siguientes conceptos que vamos a ver, son características que tiene un buen diseño[28] orientado a objetos, si bien este paradigma no garantiza dichas características, hacen mucho más fácil poder lograrlas y en cierta manera nos obliga a usarlas.

Abstracción La abstracción es la propiedad de los objetos que consiste en tener en cuenta sólo los aspectos más importantes desde un punto de vista determinado y no tener en cuenta otros aspectos. Durante el proceso de abstracción es cuando se decide qué características y comportamientos debe tener el modelo para así reducir su complejidad. De este modo, las características complejas se hacen más manejables.

> Ejemplo: En POO, podemos considerar una *Persona* como un objeto que tiene propiedades (como nombre, altura, peso, color

[28] El análisis orientado a objetos (Object-oriented Analysis OOA) es el proceso de analizar un problema, un sistema o una tarea —que alguien quiere convertir en una aplicación— e identificar los objetos y las interacciones entre esos objetos. La etapa de análisis tiene que ver con lo que se necesita hacer. La salida de la etapa de análisis es un conjunto de requisitos. Si tuviéramos que completar el análisis paso a paso, habríamos convertido una tarea, en un conjunto de requisitos. En cierto modo, el análisis es un nombre inapropiado. En su lugar, explora su entorno, manipula formas y ve dónde se podrían contener. Una mejor interpretación de la frase podría ser la exploración orientada a objetos. En desarrollo de Software, las etapas iniciales de análisis incluyen entrevistar a los clientes, estudiando sus procesos, y eliminando posibilidades.

El diseño orientado a objetos (Object-oriented Design OOD) es el proceso de convertir tales requisitos en una especificación de implementación. El diseñador debe nombrar los objetos, definir los comportamientos, y especificar formalmente qué objetos pueden activar comportamientos específicos en otros objetos. La etapa de diseño tiene que ver con cómo se deben hacer las cosas. La salida de la etapa de diseño es una especificación de implementación. Si fuéramos a completar la etapa de diseño en un solo paso, habríamos cumplido los requisitos definidos durante análisis orientado a objetos en un conjunto de clases e interfaces que podrían implementarse en (idealmente) cualquier lenguaje de programación orientado a objetos.

La programación orientada a objetos (Object-oriented Programming OOP) es el proceso de conversión del diseño en un programa de trabajo que haga exactamente lo que el solicitante del mismo originalmente solicitó. ¡Sí claro!, sería encantador si el mundo se encontrara con este ideal y pudiéramos seguir estas etapas una por una, en perfecto orden, como todos los antiguos libros de texto nos decían. Como de costumbre, el mundo real es mucho mas turbio, no importa cuánto intentemos separar estas etapas, siempre encontraremos cosas que necesitan más análisis mientras estamos diseñando. Cuando estamos programando, encontramos características que necesitan aclaración en el diseño.

> de pelo, color de ojos, etcétera) y métodos (como hablar, mirar, andar, correr, parar, etcétera). Gracias a la abstracción, el objeto *Tren* puede manipular objetos del tipo *Persona* sin tener en cuenta sus propiedades ni métodos ya que sólo le interesa, por ejemplo, calcular la cantidad de personas que están viajando en él en ese momento, sin tener en cuenta ninguna otra información relacionada con dichas personas, tales como la altura, el nombre, el color de ojos, etcétera. Nuestro objeto *Tren* se abstrae del objeto del tipo *Persona*.

Ocultamiento de la información Este concepto hace referencia a que los componentes se deben utilizar como si sólo se conociera su interfaz y no se tuviera conocimiento de su implementación. En otras palabras, un objeto sabe que otro objeto entiende un determinado mensaje, el cual recibe ciertos parámetros y devuelve algo (o no).

> Ejemplo: Yo como objeto *Médico* quiero saber cuantas cirugías tuvo un paciente, y entonces le envía un mensaje para que el *Paciente* me devuelva el número de cirugías que tuvo. El objeto *Médico* no sabe como devolvió ese número el *Paciente*, solo sabe que tiene que enviarle el mensaje y recibir un número.

Encapsulamiento Encapsulamiento es la capacidad de diferenciar qué partes de un objeto son parte de la interfaz y cuales permanecerán inaccesibles por el usuario. Son los lenguajes de programación los cuales, por medio de los modificadores de acceso, permiten indicar el modo de accesibilidad de un componente.

En POO, a la conjunción de abstracción y ocultamiento de implementación se le llama encapsulamiento.

> Según Booch, encapsulamiento "... es el proceso de almacenar en un mismo comportamiento los elementos de una abstracción que constituyen su estructura y su comportamiento; sirve para separar la interfaz contractual de una abstracción y su implementación".

Veamos a través de un ejemplo:

```
public class TestPersona {
   public static void main (String [ ] Args) {
      Persona p = new Persona ("Mauro", "Ruiz perez", 36);
      System.out.println (p.edad);
   }
}
```

En este caso, podemos acceder al valor de la *edad* del objeto p de la clase *Persona*, pero lo recomendable es mandar el mensaje que solicite dicho valor mediante *getEdad*().

Viéndolo desde el punto de vista de la escalabilidad y de un lenguaje de programación, si el día de mañana tenemos diferentes objetos que tienen que modificar dicha propiedad, nuestro código se puede volver difícil de cambiar debido a que en varias partes se usa la propiedad *edad*, si tenemos encapsulado ese comportamiento, solo vamos a tener que hacer un cambio en un solo lugar.

Cohesión y Acoplamiento La cohesión mide la relación entre el concepto que deseamos modelar y las responsabilidades del componente que lo representan. El acoplamiento mide qué tan relacionados están los componentes del sistema entre sí y cómo esa dependencia provoca un diseño. Siempre buscaremos tener un bajo acoplamiento y que los objetos sean lo más cohesivos posible.

Claramente se puede ver que ambos conceptos están inversamente relacionados, nuestros diseños deben tener una alta cohesión y un bajo acoplamiento. El paradigma de objetos se basa en esta regla para generar diseños más sencillos, y al mismo tiempo fáciles de programar, probar y mantener.

Declaratividad y Expresividad Estos conceptos no son particulares de POO, pero aun así se nos olvidan, por eso les vamos a hacer un repaso.

La expresividad tiene que ver con qué tan claro es el código, qué tanto expresa lo que tenía en en mente el programador cuando lo escribió, es decir, que tan bien expresadas están las ideas del programador. Usar variables y métodos con nombres que representan lo que son o hacen, es una forma fácil de aplicar la expresividad, que no tiene nada que ver con el paradigma sino con el programador.

Por lo general la declaratividad ayuda a que el código sea más expresivo, porque no se mezcla el algoritmo con lo que se quiere que haga más a alto nivel. Cuando tenemos un código bien declarativo, podemos entender que

es lo que hace sin ver en detalle el algoritmo implementado. Es importante recalcar que la declarativid es contraria a la imperatividad (o sea, detallar línea a línea los pasos que hace el algoritmo).

Entonces, la declarativid indica qué se hace y la expresividad muestra la intención de lo que va a hacer el código.

Recordemos una frase de Martin Fowler que vale la pena comentar:

> "Cuando sientas la necesidad de escribir un comentario, intenta primero refactorizar el código de manera que cualquier comentario se convierte en innecesario."

La expresividad y la declarativid son muy importantes para poder generar un código autodocumentado, ya que nunca sabemos en el futuro quienes van a tener que revisar nuestro código. Si un compañero que tiene que modificar nuestro código y no logra comprender fácilmente que quisimos hacer, tenemos un problema. Si nosotros volvemos a un código que escribimos y no lo entendemos, tenemos un grave problema.

3.1 Herencia y Composición

Sobrecarga de métodos La sobrecarga es la capacidad de un lenguaje de programación, que permite nombrar con el mismo identificador diferentes variables u operaciones.

La sobrecarga de métodos se refiere a la posibilidad de tener dos o más métodos con el mismo nombre pero diferentes parámetros. El compilador usará una u otra dependiendo de los parámetros usados.

El mismo método dentro de una clase permite hacer cosas distintas en función de los parámetros.

Clase Vector:

```
public class Vector {
   // Coeficientes del vector
   private double []C;
   // Guarda la dimension del arreglo de coeficientes
   private int Dim;
   // Constructor de la clase
   public Vector() {
      Dim = 0;
```

```
}
// Asigna coeficientes
public Vector(double []coef) {
   int i, n = coef.length;
   C = new double[n];
   Dim = n;
   for (i = 0; i < n; i++) C[i] = coef[i];
}
// Retorna los coeficientes
double coeficiente(int i) {
   if (i >= Dim) {
        System.out.print("Error");
        return 0;
  }
   return C[i];
}
// Retorna la dimension del arreglo de coeficientes
int dimension() {
   return Dim;
}
// Visualiza el vector
public void visualiza() {
   int i;
   System.out.print("(");
   for (i = 0; i < Dim; i++) {
      System.out.print(C[i]);
      if(i < Dim-1) System.out.print(", ");
   }
   System.out.print(")");
}
// Visualiza el vector
public void visualizaLN() {
   visualiza();
   System.out.println("");
}
// Suma con dos operandos
public void suma(Vector a, Vector b) {
   int i;
```

```
            Dim = 0;
            // Revisa el tamano de los vectores a sumar
            if(a.dimension() != b.dimension()) {
              System.out.println("Error, las dimensiones no son iguales");
            } else {
               // Solicita memoria para los coeficientes del vector
resultante
              Dim = a.dimension();
              C = new double[Dim];
              // Realiza la suma entre los coeficientes comunes
              for (i = 0; i < Dim; i++) C[i] = a.coeficiente(i) +
b.coeficiente(i);
            }
          }
          // Suma con un operando
          public void suma(Vector a) {
            int i;
            // Revisa el tamano de los vectores a sumar
            if(dimension() != a.dimension()) {
              System.out.println("Error, las dimensiones no son iguales");
            } else {
              // Realiza la suma entre los coeficientes comunes
              for (i = 0; i < Dim; i++) C[i] += a.coeficiente(i);
            }
          }
          // Funcion Principal ....
          public static void main(String[] args) {
            double a[] = {3.0, -4.0, 1.5};
            double b[] = {1.0, 1.0, 2.0};
            double c[] = {1.0, 3.0, -3.0};
            Vector A = new Vector(a);
            Vector B = new Vector(b);
            Vector C = new Vector(c);
            System.out.println("Suma con un operando:");
            C.visualizaLN();
            A.visualizaLN();
            C.suma(A);
            C.visualizaLN();
```

```
        System.out.println("Suma con dos operandos:");
        A.visualizaLN();
        B.visualizaLN();
        C.suma(A, B);
        C.visualizaLN();
    }
}
```

En este ejemplo que vemos, uno podría invocar al método "constructor" y "Suma" y según el número y tipo de parámetros o sin ellos de como se invoque en nuestro código, el compilador decidirá a cual llamar según corresponda.

En tiempo de compilación, se buscan todas las llamadas a este método y según el tipo de los parámetros con los que se esté invocando y el objeto que puede o no devolver (necesariamente son lenguajes de chequeo estático), se determina a qué implementación llamará. Si la combinación de parámetros no coincide, el compilador mandará mensaje de error.

Los multimétodos, a diferencia de la sobrecarga, son un conjunto de métodos con la misma firma -el nombre y los parámetros del método-, pero que se pueden solapar, y se decide cuál ejecutar en tiempo de ejecución.

Composición La composición se refiere a la combinación de objetos simples para hacer objetos más complejos.

Los objetos a menudo pueden dividirse en tipos compuestos y componentes, y la composición puede considerarse como una relación entre estos tipos: un objeto de un tipo compuesto (ej.: auto) "tiene un" objeto de un tipo simple (ej.: rueda).

Considere la relación de un automóvil con sus partes: el automóvil tiene o se compone de objetos como el volante, asientos, caja de cambios y el motor. Esta relación podría definirse como una relación de composición.

Los objetos compuestos generalmente se expresan por medio de referencias de un objeto a otro. Tales referencias pueden ser conocidas como atributos, campos, miembros o propiedades y la composición resultante como tipo compuesto. Sin embargo, tener esas referencias no necesariamente significa que un objeto es compuesto. Sólo se llama compuesto, si los objetos de los que se compone son realmente sus partes, es decir, no tienen existencia independiente.

Herencia Es el mecanismo por el cual un objeto se basa en otro objeto o clase, extendiendo la implementación para reutilizar su comportamiento[29].

Supongamos que la clase Médico y Profesor, por ser Personas, comparten datos personales, esto queda plasmado en la siguiente jerarquía de clases:

Clase Medico:

```
public class Medico extends Persona {
  private String especialidad;
  private String cedula;
  //Constructor de la subclase: incluimos como parametros
al menos los del constructor de la superclase
  public Medico(String nombre, String apellidos, int edad,
String especialidad, String cedula) {
    super(nombre, apellidos, edad);
    this.especialidad = especialidad;
    this.cedula = cedula;
  }
  public void setEspecialidad(String especialidad) {
    this.especialidad = especialidad;
  }
  public String getEspecialidad() {
    return especialidad;
  }
  public void setCedula(String cedula) {
```

[29] En Java no se permite herencia múltiple, pero sí en Paython y C++, permitiendo diseños más complejos.

- La herencia múltiple entra en escena cuando una clase hereda más de una clase base. Ejemplo: una clase que define un hijo hereda de dos clases base, Madre y Padre.
- La herencia multinivel significa que una clase hereda de otra clase que a su vez es una subclase de alguna otra clase base. Ejemplo: una clase que describe un automóvil deportivo heredará de una clase base Auto, que a su vez hereda otra clase Vehículo.
- La herencia híbrida es una combinación de herencia múltiple y de varios niveles.
- La herencia jerárquica se refiere a la herencia en la que una clase base tiene más de una subclase. Por ejemplo, la clase de vehículo puede tener "coche", "bicicleta", etc. como subclases.

```
        this.cedula = cedula;
      }
      public String getCedula() {
        return cedula;
      }
      public void visualiza() {
        System.out.println ("Nombre del medico: " + getNom-
bre() + " " + getApellidos() +
    " especialidad: " + getEspecialidad() + " cedula: " + getCedula());
      }
    }
```

Clase Profesor:

```
    public class Profesor extends Persona {
      private String idProfesor;
      //Constructor de la subclase: incluimos como parametros
al menos los del constructor de la superclase
      public Profesor(String nombre, String apellidos, int edad,
String idProfesor) {
        super(nombre, apellidos, edad);
        this.idProfesor = idProfesor;
      }
      public void setIdProfesor (String idProfesor) {
        this.idProfesor = idProfesor;
       }
      public String getIdProfesor () {
        return idProfesor;
      }
      public void visualiza() {
        System.out.println ("Nombre del profesor: " + getNom-
bre() + " " + getApellidos() +
    " Id de profesor: " + getIdProfesor() );
      }
    }
```

¿Cómo funciona?

```
public class TestHerencia {
  public static void main (String [ ] Args) {
    Profesor p1 = new Profesor ("Juan", "Nadrie Garcia",
33, "Prof 22-387-11");
    Medico m1 = new Medico ("Roberto", "Gonzalez Gar-
cia", 33, "Cirujano", "328943784");
    Persona p = new Persona ("Mauro", "Ruiz perez", 36);
    p.visualiza();
    p1.visualiza();
    m1.visualiza();
  }
}
```

Cuando le llega un objeto, el compilador, hace lo siguiente:

1. Busca la implementación en la clase del objeto a la que le llega el mensaje
2. Si no la encontra, la busca en su padre
3. Repite el paso anterior hasta que no haya más padres.
4. Si no encuentra ninguna implementación en todos los niveles, lanza un error.

A este mecanismo se lo conoce como **Method LookUp**. Algo a destacar de la herencia es que las clases hijas van a heredar lo que haya definido su superclase, tanto atributos como métodos.

Los motivaciones de utilizar herencia son:

- Reusa código.
- Genera una abstracción.
- Crea un Tipo (para lenguajes con tipado estático).

Prototipos La orientación a objetos basada en prototipos es un estilo de reutilización de comportamiento (herencia) que se logra por medio de la clonación de un objeto ya existente, que sirve como prototipo (Javascript, es el lenguaje orientado a objetos basado en prototipos más conocido).

Autoreferencia Así como un objeto puede conocer a otro objeto teniendo una referencia hacia este, también se puede conocer a sí mismo. Cualquier objeto tiene una autoreferencia, denominada *this* o *self* (según el lenguaje) para poder mandarse algún mensaje a sí mismo.

Así como el *self*, hay otra referencia, denominada *super* o *parent* la cual es equivalente al *this* o *self*, con la particularidad de que le dice al *Method LookUp* "empieza desde uno más arriba". Es decir, que no busca la implementación en la clase dónde está ejecutando el método, sino en la inmediata superior (puede que eso desemboque en que siga subiendo niveles). En el 99% de los casos, sólo deberíamos llamar a *super/parent* para ejecutar el mismo método que estamos ejecutando.

Ejemplo de código:

```
public Profesor (String nombre, String apellidos, int edad
  , String idProfesor) {
  super(nombre, apellidos, edad);
  this.idProfesor = idProfesor;
}
```

Contratos

Diseño por contratos El diseño por contratos asume que todos los componentes del cliente que invocan una operación en un componente del servidor van a encontrar las precondiciones (y postcondiciones) especificadas como obligatorias para esa operación.

Muchas veces podemos tener algunas cosas para validar en nuestro sistema. Esto es:

- Pre condiciones
- Post condiciones
- Condiciones "permanentes" o invariantes

Por ejemplo: yo siempre que alimente a mi mascota, debo garantizar que tenga hambre. Por lo que quisiera tener algo como:

```
public class Mascota implements Domesticable {
  public void alimentarse(){
    if( tengoHambre() == true) {
      //ejecutar método
    }
  }
}
```

Lo mismo puede suceder con las post condiciones, o condiciones que se deben dar en todo momento.

```
public class Mascota implements Domesticable {
  public void alimentarse(){
    if( tengoHambre() == true) {
      //ejecutar método
      estoyLleno()
    }
  }
}
```

Para hacer cumplir estos contratos, hay 2 maneras:

1. Validación manual y lanzamiento de excepciones
2. Integrado por el lenguaje o algún Framework

Lo importante es que se entienda el concepto del contrato, para poder interactuar de cierta manera con un componente, debo cumplir ciertos requisitos, tanto antes, durante o después de la interacción.

Contratos y Herencia Básicamente, aplican las mismas ideas y conceptos que en la herencia de comportamiento. Las precondiciones, postcondiciones e invariantes se heredan de clases a subclases.

Sin embargo existen algunos condicionamientos, para garantizar el principio de intercambiabilidad. Que se pueden resumir en la siguiente frase:

Require no more, and promise no less

Relacionado con la L (Liskov Substitution Principle) de SOLID[30], la cual plantea que donde uso una clase, debería poder usar cualquier subclase de ella y que siga funcionando todo correctamente.

[30]SOLID Design Principles de Robert C. Martin's:

Tipos ¿Qué es un tipo? Un tipo describe un conjunto de valores.
La idea de tipo nos permite relacionar:

- Un conjunto de valores que tienen ese tipo o son de ese tipo
- Las operaciones que pueden ser realizadas sobre esos valores.

Los objetivos de un sistema de tipos son:

- Ayudar a detectar errores al programar.
- Guiar al programador sobre las operaciones válidas en un determinado contexto, tanto la documentación como en las ayudas automáticas que puede proveer por ejemplo un IDE.
- En algunos casos el comportamiento de una operación puede variar en función del tipo de los elementos involucrados en la misma. Polimorfismo, sobrecarga (los veremos más adelante), multimétodos, etc.

Podemos hacer 3 clasificaciones de tipado:

- El implícito o explícito. Un lenguaje va a tener tipado explícito sí:

Todos los elementos (variables, métodos, etc.) tienen un tipo definido.

Para que dos objetos sean polimórficos, debo explicitarlos (por una interfaz o por heredar de la misma clase).

- Chequeo dinámico o estático.

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion

https://es.wikipedia.org/wiki/SOLID

La diferencia está en el momento en que es ejecutado el chequeo de tipos. En los lenguajes de chequeo estático (como Java o C++) se verifica en tiempo de compilación, mientras que en los de chequeo dinámico se realiza en tiempo de ejecución (como en Python).

El chequeo dinámico da lugar a un concepto famoso denominado Duck Typing. El cual se basa en que "*si algo tiene pico de pato, camina como pato, y hace cuack, es un pato*". Es decir, toma sentido el tipo al que pertenece un objeto según qué mensaje puedo mandarle y cómo responde, no se me especifica desde antes.

- Estructural o nominal

Hace referencia a si se identifica un tipo por su nombre o por su estructura. Es muy común en los lenguajes del paradigma funcional.

Las combinaciones más frecuentes son:

- Explícito, estático y nominal
- Implícito y dinámico

Casting Es un mecanismo utilizado en los lenguajes de chequeo estático, mediante el cual le "aseguramos" al compilador que cierto objeto pertenece al tipo especificado.

Ejemplo:

```
import java.util.*;
public class TestHerencia {
   public static void main (String [ ] Args) {
      Profesor p1 = new Profesor ("Juan", "Nadrie Garcia",
33, "Prof 22-387-11");
      Medico m1 = new Medico ("Roberto", "Gonzalez Gar-
cia", 33, "Cirujano", "328943784");
      Persona p = new Persona ("Mauro", "Ruiz Perez", 36);
      List<Persona> tlist = new ArrayList<Persona>();
      tlist.add(p1);
      tlist.add(m2);
```

```
        tlist.add(p);
        for (int i = 0; i < tlist.size(); i++) tlist.get(i).visualiza();
    }
}
```

Hay que ser muy cuidadosos al usar el casteo, porque podría llevar a "confundir" al compilador y hacer que falle en tiempo de ejecución (perdiendo el beneficio que nos da el chequeo estático), por ejemplo si en el *objetotlist* que recibe objetos de las clases *Medico, Profesor y Persona*, al llamar a métodos no comunes indicados en la clase base *Persona* nuestro programa lanzará una excepción.

Variables y Métodos de Clase También se denominan métodos y variables estáticas. La implementación de los mismos varía dependiendo del lenguaje. En algunos las clases son objetos, en otros, las clases son entes particulares con una instancia asociada, que se pueden utilizar de determinada forma y enviarles mensajes y en algunos no se utiliza ninguna de estas nociones, pero siguen teniendo el mismo concepto.

Las variables de clase nos sirven cuando queremos que nuestros objetos tengan alguna referencia a algún valor, que sea el mismo para todas las instancias de esa clase, y que además pueda cambiar (por eso usamos una variable y no forzamos ese valor en el código del programa, asumiendo que es posible).

Si usáramos una variable de instancia con la intención de definir el mismo valor en todas las instancias de esa clase, y luego queremos cambiar dicho valor, debemos poder encontrar todas las instancias ya existentes para poder mandarles el mensaje para que actualicen su referencia. Y si tenemos muchos objetos, podemos tener un gran problema, no sólo por la complejidad innecesaria del problema, sino también porque el desempeño se vera afectado.

Los métodos de clase, son métodos cuyo receptor / implementador va a ser una clase en lugar de un objeto. El caso más común es el del constructor, donde la clase recibe el "New" y ella sabe generar una nueva instancia de la clase e incluso llama al constructor correspondiente.

Otro caso recurrente es el de querer tener una sola instancia por clase. Para profundizar más en este funcionamiento, ver el patrón de diseño Singleton.

Herencia V.S. Composición Uno de los objetivos que buscamos cuando programamos es reutilizar métodos y funcionalidades, para lograr una mayor mantenibilidad. Dentro de los lenguajes convencionales orientados a objetos existen varias formas de hacer esto, las dos más conocidas son: herencia de clases y composición.

Herencia de clases También conocido como "reutilización de caja blanca", debido a la visibilidad que dan ya que mediante la herencia, las implementaciones de las clases padres se hacen visible a las clases hijas.

Ventajas:

- Se define en tiempo de compilación y no tiempo de ejecución, es decir que conocemos con seguridad que se va a ejecutar.
- Es más sencillo modificar la implementación que está siendo reutilizada.

Desventajas:

- Como dijimos antes, se definen en tiempo de compilación, por lo que no se puede cambiar el comportamiento de un objeto en tiempo de ejecución, lo cual puede ser un requisito de nuestro diseño.
- Se rompe el concepto de encapsulación al exponer los detalles de la implementación en la clase padre.

Composición Conocida como "reutilización de caja negra", a diferencia del caso anterior nosotros no conocemos los detalles de la implementación, ya que la misma se encuentra encapsulada en el objeto al cual estamos invocando.

Ventajas:

- Al tener objetos que hacen referencia a otros objetos, el vínculo se define en tiempo de ejecución y como a dichos objetos se accede mediante su interfase no se rompe el principio de encapsulación (entonces podríamos reemplazar al objeto por otro cuando corre nuestro programa y tener un comportamiento totalmente distinto).
- Permite tener clases más centradas y encapsuladas, haciendo que nuestro diseño tenga más objetos (menos clases) de menor tamaño y con menos responsabilidades.

Agregación Es cuando una o más clases son parte de otra clase. En este caso el objeto no es creado dentro del objeto compuesto, es adherido mediante algún método que lo permita.

Diferencia entre Agregación y Composición

- Las relaciones en una composición son requeridas, en la agregación son opcionales.
- En la composición una clase particular no puede ser compartida por otras clases compuestas, en la agregación esto es posible.
- La relación de vida de la clase particular y la clase contenedora, es muy fuerte, de hecho es la relación más fuerte; tanto que si un objeto de la clase contenedora es destruido la clase particular también lo será. Esto en la agregación no ocurre.

Conclusión Se podría decir que conviene favorecer la composición sobre la herencia, sin embargo esta respuesta no es definitiva, cada uno tiene que identificar la necesidades y el dominio en el cual está trabajando para identificar en que casos conviene usar cada metodología.

Algunas recomendaciones:

- En lenguajes que no soportan herencia múltiple, puede resultar ventajosa la composición.
- La composición genera un diseño más desacoplado, que puede ayudar a hacer pruebas individuales de las clases y de la interacción de los objetos de una más fácil.
- Muchos patrones de diseño favorecen la composición.

3.2 Clase Abstracta

Las clases abstractas son aquellas que por sí mismas no se pueden identificar como algo "concreto" (no existen como tal en el mundo real), pero sí poseen determinadas características que son comunes a otras clases que pueden ser creadas a partir de ellas.

Clase Numero:

```
public abstract class Numero {
   // Genera un nuevo miembro de la clase
   abstract Numero nuevo();
   // Regresa verdadero si es cero
   abstract boolean esCero();
   // Suma
   abstract void suma(Numero a, Numero b);
   // Suma
   abstract void suma(Numero a);
   // Visualiza sin cambiar de linea
   abstract void visualiza();
   // Visualiza cambiando de linea
   void visualizaLN() {
   visualiza();
     System.out.println("");
   }
}
```

Con ella definimos la clase NumeroFraccionario:

```
public class Fraccion extends Numero {
   // Numerador de la fraccion
   private long P;
   // Denominador de la fraccion
   private long Q;
   // Arreglo que contendra a los primos
   static private long []Pr;
   // Numero de primos que contendra el arreglo de primos
   static final private int nPr = 1000;
   // Constructor nulo fraccion 0/1
   public Fraccion() {
     this(0,1);
   }
   // Constructor con un objeto Fraccion
   public Fraccion (Fraccion a) {
     this(a.numerador(), a.denominador());
   }
   // Constructor con entero P Fraccion P/1
```

```
public Fraccion (long p) {
  this(p,1);
}
// Costructor con la fraccion P/Q
public Fraccion (long p, long q) {
  P = p;
  Q = q;
  calculaPrimos();
}
// Genera un nuevo miembro de la clase
public Numero nuevo() {
  Numero c = new Fraccion();
  return c;
}
// Calcula los primeros nPr primos
private void calculaPrimos() {
  int n, i, np;
  Pr = new long[nPr];
  // Guarda los primeros 2 primos
  Pr[0] = 2;
  Pr[1] = 3;
  np = 2;
  // Empieza la busqueda de primos a partir de 4
  n = 4;
  // Ciclo para buscar los primeros NPB primos
  while (np < nPr) {
    for (i = 0; i < np; i++) {
      if((n % Pr[i]) == 0) break;
    }
    if(i == np) {
      Pr[i] = n;
      np++;
    }
    n++;
  }
}
// Visualiza los primos encontrados
public void visArregloPrimos() {
```

```
            System.out.println("Visualiza los primeros " + nPr + "
    primos");
            for (int i = 0; i < nPr; i++) System.out.println(Pr[i]);
          }
          // Simplifica el numerador y denominador de la fraccion
          private void simplifica() {
            int i, sw;
            for (i = 0; i < nPr; i++) {
              do {
                sw = 0;
                if (Math.abs(P) < 2 || Q < 2) return;
                if((P % Pr[i] == 0) && (Q % Pr[i] == 0)) {
                  P /= Pr[i];
                  Q /= Pr[i];
                  sw = 1;
                }
              } while(sw == 1);
            }
          }
          // Retorna el Numerador
          public long numerador() {
            return P;
          }
          // Retorna el denominador
          public long denominador() {
            return Q;
          }
          // Regresa verdadero si la fraccion es cero
          public boolean esCero() {
            if (numerador() == 0) return true;
            return false;
          }
          // Suma de fracciones con dos operandos
          public void suma(Numero A, Numero B) {
            Fraccion a = new Fraccion( (Fraccion) A);
            Fraccion b = new Fraccion( (Fraccion) B);
            P = a.numerador() * b.denominador() + b.numerador()
    * a.denominador();
```

```
            Q = a.denominador() * b.denominador();
        }
        // Suma de fracciones con un operando
        public void suma(Numero A) {
            Fraccion a = new Fraccion( (Fraccion) A);
            P = numerador() * a.denominador() + a.numerador() *
denominador();
            Q = denominador() * a.denominador();
        }
        // Visualiza la Fraccion
        public void visualiza() {
            simplifica();
            System.out.print(P);
            System.out.print("/");
            System.out.print(Q);
        }
    };
```

También definimos la clase NumeroComplejo:

```
    public class Complejos extends Numero {
        // Parte real del complejo
        private double R;
        // Parte imaginaria del complejo
        private double I;
        // Constructores
        public Complejos() {
            R = 0;
            I = 0;
        }
        // Constructores
        public Complejos(double r, double i) {
            R = r;
            I = i;
        }
        // Constructores
        public Complejos(double r) {
            R = r;
```

```
            I = 0;
          }
          // Constructores
          public Complejos(Complejos a) {
            R = a.parteReal();
            I = a.parteImaginaria();
          }
          // Genera un nuevo miembro de la clase
          public Numero nuevo() {
            Numero c = new Complejos();
            return c;
          }
          // Retorna la parte real del complejo
            public double parteReal() {
            return R;
          }
          // Retorna la parte imaginaria del complejo
          public double parteImaginaria() {
            return I;
          }
          // Regresa verdadero si el complejo es cero
           public boolean esCero() {
           if (parteReal() == 0.0 && parteImaginaria() == 0.0)
    return true;
            return false;
          }
          // Suma con dos operandos
          public void suma(Numero A, Numero B) {
            Complejos a = new Complejos( (Complejos) A);
            Complejos b = new Complejos( (Complejos) B);
            R = a.parteReal() + b.parteReal();
            I = a.parteImaginaria() + b.parteImaginaria();
         }
          // Suma con un operando
          public void suma(Numero A) {
            Complejos a = new Complejos( (Complejos) A);
            R = parteReal() + a.parteReal();
            I = parteImaginaria() + a.parteImaginaria();
```

```
    }
    // Visualiza el complejo
    public void visualiza() {
      System.out.print(R + " + " + I + "i");
    }
};
```

Y también definimos la clase NumeroComplejoFraccionario:

```
public class ComplejoFraccionario extends Numero {
    // Parte real del complejo
    private Fraccion R;
    // Parte imaginaria del complejo
    private Fraccion I;
    // Constructores
    public ComplejoFraccionario() {
      R = new Fraccion();
      I = new Fraccion();
    }
    // Constructores
    public ComplejoFraccionario(long r, long i) {
      R = new Fraccion(r);
      I = new Fraccion(i);
    }
    // Constructores
    public ComplejoFraccionario(long r) {
      R = new Fraccion(r);
      I = new Fraccion();
    }
    public ComplejoFraccionario(long rn, long rd, long in, long
id) {
      R = new Fraccion(rn, rd);
      I = new Fraccion(in, id);
    }
    // Constructores
    public ComplejoFraccionario(ComplejoFraccionario a) {
      R = a.parteReal();
      I = a.parteImaginaria();
```

```
        }
        // Genera un nuevo miembro de la clase
          public Numero nuevo() {
          Numero c = new ComplejoFraccionario();
          return c;
        }
        // Retorna la parte real del complejo
        public Fraccion parteReal() {
          return R;
        }
        // Retorna la parte imaginaria del complejo
        public Fraccion parteImaginaria() {
          return I;
        }
        // Regresa verdadero si el complejo es cero
        public boolean esCero() {
          if (parteReal().numerador() == 0.0 && parteImaginaria().numerador()
== 0.0) return true;
          return false;
        }
        // Suma con dos operandos
        public void suma(Numero A, Numero B) {
           ComplejoFraccionario a = new ComplejoFraccionario(
(ComplejoFraccionario) A);
           ComplejoFraccionario b = new ComplejoFraccionario(
(ComplejoFraccionario) B);
          R.suma(a.parteReal(), b.parteReal());
          I.suma(a.parteImaginaria(), b.parteImaginaria());
        }
        // Suma con un operando
        public void suma(Numero A) {
           ComplejoFraccionario a = new ComplejoFraccionario(
(ComplejoFraccionario) A);
          R.suma(parteReal(), a.parteReal());
          I.suma(parteImaginaria(), a.parteImaginaria());
        }
        // Visualiza el complejo
        public void visualiza() {
```

```
            R.visualiza();
            System.out.print("+");
            I.visualiza();
            System.out.print("i");
        }
    };
```

Pero, ¿podemos definir algún objeto *Número* o todos van a ser *Fraccionario/Complejo/ComplejoFraccionario*/etc.?

En este caso, la clase *Número* no debería ser instanciable, solo quiero utilizarla para que las clases que heredan de esta puedan usar el código definido en ella.

A esto se le llama una **clase abstracta**. Al contrario de las abstractas, las clases que sí son instanciables, se les denomina concretas. Una clase abstracta define sólo la interfaz o firma -el nombre, los parámetros del método y lo que retorna- de algunos de sus métodos. Podemos tener:

- Clases abstractas puras, (véase ejemplo en 3.5.1).
- Clases parcialmente abstractas, (véase ejemplo en 3.2).

Manteniendo el concepto, una clase abstracta puede obligar a que sus subclases implementen cierto método, pero sin definir ningún comportamiento por omisión. A estos métodos, naturalmente, se los denomina métodos abstractos.

Si una clase tiene al menos un método abstracto, entonces debe ser abstracta, porque no tiene sentido que haya un objeto que sea instancia de ella. A fines de comprensión de concepto, podría decirse que una interfaz es una clase abstracta con todos sus métodos abstractos.

Interfaces Una interfaz define un tipo y es una colección de métodos abstractos. De forma similar a una clase abstracta, obliga a quienes implementen la interfaz a implementar los métodos.

Las interfaces no pueden definir la implementación de los métodos de los objetos que las van a utilizar, pero si obligan a dichos objetos a definir esa implementación.

Las interfaces sirven para solventar la limitación de muchos lenguajes, que no soportan la herencia múltiple.

Clases Abstractas Puras vs Interfaces

- Solo se puede extender de una clase abstracta, pero se pueden implementar más de una interfaz en una clase.
- Una interfaz no puede definir atributos.
- Así como no se puede crear instancias de clases abstractas, tampoco es posible crear instancias de interfaces.

3.3 Polimorfismo y Programación Genérica

Es la capacidad que tiene un objeto de poder tratar indistintamente a otros que sean potencialmente distintos, es decir, es la capacidad que tienen distintos objetos de entender un mismo mensaje.

Clase Matriz:

```
public class Matriz <T extends Numero> {
   // Coeficientes de la Matriz
   private Numero [][] M;
   // Numero de Renglones
   private int ren;
   // Numero de Columnas
   private int col;
   // Constructor de la clase
   public Matriz() {
      col = 0;
      ren = 0;
   }
   // Asigna coeficientes
   public void asignaCoeficientes(Numero [][]coef) {
      int i,j;
      ren = coef.length;
      col = coef[0].length;
      M = new Numero[ren][col];
      for (i = 0; i < ren; i++) {
         for (j = 0; j < col; j++) {
            M[i][j] = coef[i][0].nuevo();
            M[i][j] = coef[i][j];
```

```
      }
    }
  }
  // Retorna los coeficientes
  Numero coeficiente(int i, int j) {
    if (i < 0 || i >= ren || j < 0 || j >= col) System.out.print("Error");
    return M[i][j];
  }
  // Retorna el numero de renglones
  int renglones() {
    return ren;
  }
  // Retorna el numero de columnas
  int columnas() {
    return col;
  }
  // Visualiza la Matriz
  public void visualiza() {
    int i, j;
    for (i = 0; i < ren; i++) {
      for (j = 0; j < col; j++) {
        M[i][j].visualiza();
        System.out.print(" ");
      }
      System.out.println("");
    }
  }
  // Visualiza la Matriz
  public void visualizaLN() {
    visualiza();
    System.out.println("");
  }
  // Suma con dos operandos
  public void suma(Matriz a, Matriz b) {
    int i, j;
    ren = 0;
    col = 0;
    // Revisa el tamano de las Matrices a sumar
```

```
            if(a.renglones() != b.renglones() || a.columnas() != b.columnas())
{
               System.out.println("Error, las dimensiones no son iguales");
            } else {
               // Solicita memoria para la Matriz resultante
               ren = a.renglones();
               col = a.columnas();
               M = new Numero[ren][col];
               for (i = 0; i < ren; i++) {
                  for (j = 0; j < col; j++) {
                     M[i][j] = a.coeficiente(i,j).nuevo();
                     M[i][j].suma(a.coeficiente(i,j), b.coeficiente(i,j));
                  }
               }
            }
         }
         // Suma con un operando
         public void suma(Matriz a) {
            int i, j;
            ren = 0;
            col = 0;
            // Revisa el tamano de las Matrices a sumar
            if(renglones() != a.renglones() || columnas() != a.columnas())
{
               System.out.println("Error, las dimensiones no son iguales");
            } else {
               // Solicita memoria para la Matriz resultante
               for (i = 0; i < ren; i++) {
                  for (j = 0; j < col; j++) {
                     M[i][j].suma(a.coeficiente(i,j));
                  }
               }
            }
         }
         // Funcion Principal ....
         public static void main(String[] args) {
            // Ejemplito de Matrices de coeficientes Complejos
            Complejos [][]x = new Complejos[2][2];
```

```
        x[0][0] = new Complejos(2, 3);
        x[1][0] = new Complejos(2, 2);
        x[0][1] = new Complejos(3, 2);
        x[1][1] = new Complejos(3, 2);
        Complejos [][]y = new Complejos[2][2];
        y[0][0] = new Complejos(-2, -3);
        y[1][0] = new Complejos(2, 3);
        y[0][1] = new Complejos(3, 3);
        y[1][1] = new Complejos(3, 3);
        Matriz<Complejos> X = new Matriz<Complejos>();
        X.asignaCoeficientes(x);
        Matriz<Complejos> Y = new Matriz<Complejos>();
        Y.asignaCoeficientes(y);
        Matriz<Complejos> Z = new Matriz<Complejos>();
        System.out.println("Complejo");
        X.visualizaLN();
        System.out.println("+");
        Y.visualizaLN();
        Z.suma(X, Y);
        System.out.println("======================");
        Z.visualizaLN();
        System.out.println("");
        System.out.println("");

        // Ejemplito de Matrices de coeficionets ComplejoFrac-
    cionario
        ComplejoFraccionario [][]xx = new ComplejoFraccionario[2][2];
        xx[0][0] = new ComplejoFraccionario(2, 3, 4, 5);
        xx[1][0] = new ComplejoFraccionario(2, 2, 3, 7);
        xx[0][1] = new ComplejoFraccionario(3, 2);
        xx[1][1] = new ComplejoFraccionario(3, 2);
        ComplejoFraccionario [][]xy = new ComplejoFraccionario[2][2];
        xy[0][0] = new ComplejoFraccionario(-2, 3, 5, 6);
        xy[1][0] = new ComplejoFraccionario(2, 3, 7, 9);
        xy[0][1] = new ComplejoFraccionario(3, 3);
        xy[1][1] = new ComplejoFraccionario(3, 3);
        Matriz<ComplejoFraccionario> xX = new Matriz<ComplejoFraccionario>();
        xX.asignaCoeficientes(xx);
```

```
        Matriz<ComplejoFraccionario> xY = new Matriz<ComplejoFraccionario>();
        xY.asignaCoeficientes(xy);
        Matriz<ComplejoFraccionario> xZ = new Matriz<ComplejoFraccionario>();
        System.out.println("Complejo Fraccionario");
        xX.visualizaLN();
        System.out.println("+");
        xY.visualizaLN();
        xZ.suma(xX, xY);
        System.out.println("======================");
        xZ.visualizaLN();
        System.out.println("");
        System.out.println("");
    }
}
```

Este concepto se conoce como polimorfismo. Es la capacidad de intercambiar un objeto con otro, abstrayendo al que se conoce desde su implementación.

¿Por qué es importante diseñar polimórficamente? Para ganar extensibilidad. Así, al implementar otra clase que extienda a la de tipo *Numero*, no necesitaremos modificar la clase *Matriz*, es suficiente con crearle una implementación y encajará perfectamente en el sistema.

Entonces, para hacerlo genérico, se establece un contrato: "Todas las clases de tipo *Numero* deben entender el mensaje de *sumar*".

En general, un contrato establece un acuerdo entre dos (o más) partes. Si lo cumplimos, el sistema va a tener una funcionalidad o comportamiento definido. De alguna forma regula la interacción entre dos módulos de nuestra aplicación. Entonces intercambiando un módulo por otro que cumple el mismo contrato debería ser transparente para el otro módulo.

La parte del contrato más frecuentemente usada, va a ser lo que denominamos interfaz. La interfaz de un objeto es el conjunto de mensajes que entiende.

Adicionalmente, están las construcciones específicas denominadas interfaces. Estas, no son más que una especificación del conjunto de mensajes que tienen que cumplir aquellos que la implementen.

Una buena regla para identificar la necesidad de una interfaz es establecer un adjetivo común a los objetos que quiero que la implementen (*sumar/*

restar/ dividir/ multiplicar), a diferencia de una superclase, las cuales suelen ser sustantivos (*Animal/ Persona/ Auto*).

Aprovechando todo lo desarrollado hasta aquí, mostramos un par de ejemplos de programación genérica, mediante el ejemplo de la clase Vector y la clase Polinomio genérico y que soporta los tipos de números que extienden a la clase *Numero* desarrollados anteriormente y otros más.

3.4 Excepciones

¿Cuántas veces al trabajar con un programa de cómputo, este nos manda algún mensaje de error poco descriptivo?, esto probablemente se deba a un bajo nivel de detalle en el manejo de excepciones dentro de nuestro programa. El manejo de excepciones consiste en controlar los errores que surjan dentro de nuestro programa para poder tratarlos debidamente.

Tratarlos debidamente implica:

- Mostrar un mensaje más amigable a los usuarios (la más común).
- Revertir (Rollback) alguna transacción.
- Escribir en un archivo tipo registro (Log) el error para analizarlo.
- Si el error no afecta el flujo de trabajo: contenerlo para que el flujo siga.

En Java los errores en tiempo de ejecución (cuando se esta ejecutando el programa) se denominan excepciones, y esto ocurre cuando se produce un error en alguna de las instrucciones de nuestro programa, como por ejemplo cuando se hace una división entre cero, cuando un objeto es *null* y no puede serlo, cuando no se abre correctamente un archivo, etc. Cuando se produce una excepción se muestra en la pantalla un mensaje de error y finaliza la ejecución del programa.

En Java (al igual que en otros lenguajes de programación), existen mucho tipos de excepciones. En lo referente a las excepciones hay que decir que se aprenden a base experiencia, de encontrarte con ellas y de saber solucionarlas. Cuando en Java se produce una excepción se crear un objeto de una determina clase -dependiendo del tipo de error que se haya producido-, que mantendrá la información sobre el error producido y nos proporcionará

los métodos necesarios para obtener dicha información. Estas clases tienen como clase padre la clase *Throwable*, por tanto se mantiene una jerarquía en las excepciones.

A continuación mostramos algunas de las clases para que nos hagamos una idea de la jerarquía que siguen las excepciones, pero existen muchísimas más excepciones que las que mostramos:

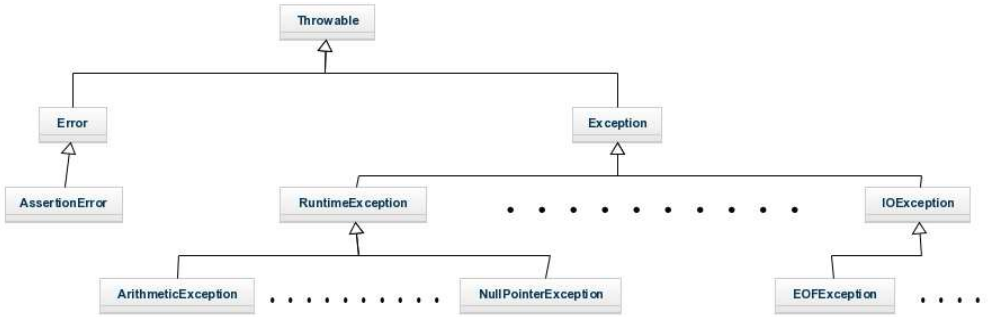

Figura 1: Jerarquía de clases para el manejo de excepciones en Java.

En Java, cuando se produce un error en un método, "se lanza" un objeto *Throwable*, cualquier método que haya llamado al método puede "capturar la excepción" y tomar las medidas que estime oportunas. Tras capturar la excepción, el control no vuelve al método en el que se produjo la excepción, sino que la ejecución del programa continúa en el punto donde se haya capturado la excepción.

Consecuencia:

> Nunca más tendremos que preocuparnos de "diseñar" códigos de error.

Throwable Clase base que representa todo lo que se puede "lanzar" en Java, contiene una instantánea del estado de la pila en el momento en el que se creó el objeto ("*Stack Trace*" o "*Call Chain*"), también almacena un mensaje (variable de instancia de tipo *String*) que podemos utilizar para detallar qué error se produjo. Además puede tener una causa, también de tipo *Throwable*, que permite representar el error que causó este error.

Error Subclase de *Throwable* que indica problemas graves que una aplicación no debería intentar solucionar (documentación de Java), ejemplos:

Memoria agotada, error interno de la JVM, etc.

Exception *Exception* y sus subclases indican situaciones que una aplicación debería tratar de forma razonable.

Los dos tipos principales de excepciones son:

- *RuntimeException*, errores del programador, como una división por cero o el acceso fuera de los límites de un array.
- *IOException*, errores que no puede evitar el programador, generalmente relacionados con la entrada/salida del programa.

Captura de excepciones: Bloques *try...catch* Se utilizan en Java para capturar las excepciones que se hayan podido producir en el bloque de código delimitado por *try* y *catch*. En cuanto se produce la excepción, la ejecución del bloque *try* termina y la cláusula *catch* recibe como argumento un objeto *Throwable*.

Ejemplo básico:

```
// Bloque 1
try {
  // Bloque 2
} catch (Exception error) {
  // Bloque 3
}
// Bloque 4
```

Flujo de la ejecución en el programa:

Sin excepciones: 1 -> 2 -> 4

Con una excepción en el bloque 2: 1 -> 2(excepción) -> 3 -> 4

Con una excepción en el bloque 1: 1(excepción)

Otro ejemplo:

```
// Bloque 1
try {
  // Bloque 2
} catch (ArithmeticException ae) {
  // Bloque 3
} catch (NullPointerException ne) {
  // Bloque 4
}
// Bloque 5
```

Flujo de la ejecución en el programa:

Sin excepciones: 1 -> 2 -> 5

Excepción de tipo aritmético: 1 -> 2(excepción) -> 3 -> 5

Acceso a un objeto nulo (null): 1 -> 2(excepción) -> 4 -> 5

Excepción de otro tipo diferente: 1 -> 2(excepción)

Un ejemplo más:

```
// Bloque1
try {
  // Bloque 2
} catch (ArithmeticException ae) {
  // Bloque 3
} catch (Exception error) {
  // Bloque 4
}
// Bloque 5
```

Flujo de la ejecución en el programa:

Sin excepciones: 1 -> 2 -> 5

Excepción de tipo aritmético: 1 -> 2(excepción) -> 3 -> 5

Excepción de otro tipo diferente: 1 -> 2(excepción) -> 4 -> 5

¡Ojo! Las cláusulas *catch* se comprueban en orden, ejemplo:

```
// Bloque1
try {
  // Bloque 2
} catch (Exception error) {
  // Bloque 3
} catch (ArithmeticException ae) {
  // Bloque 4
}
// Bloque 5
```

Flujo de la ejecución en el programa:

Sin excepciones: 1 -> 2 -> 5

Excepción de tipo aritmético: 1 -> 2(excepción) -> 3 -> 5

Excepción de otro tipo diferente: 1 -> 2(excepción) -> 3 -> 5

¡ El bloque 4 nunca se llegará a ejecutar !

La cláusula *finally* En ocasiones, nos interesa ejecutar un fragmento de código independientemente de si se produce o no una excepción (por ejemplo, cerrar un archivo que estemos manipulando).

Ejemplo:

```
// Bloque1
try {
  // Bloque 2
} catch (ArithmeticException ae) {
  // Bloque 3
} finally {
  // Bloque 4
}
// Bloque 5
```

Flujo de la ejecución en el programa:

Sin excepciones: 1 -> 2 -> 4 -> 5

Excepción de tipo aritmético: 1 -> 2(excepción) -> 3 -> 4 -> 5

Excepción de otro tipo diferente: 1 -> 2(excepción) -> 4

Si el cuerpo del bloque *try* llega a comenzar su ejecución, el bloque *finally* siempre se ejecutará:

- Detrás del bloque *try* si no se producen excepciones
- Después de un bloque *catch* si este captura la excepción.
- Justo después de que se produzca la excepción si ninguna cláusula *catch* captura la excepción y antes de que la excepción se propague hacia arriba.

Lanzamiento de Excepciones La sentencia *throw* se utiliza en Java para lanzar objetos de tipo *Throwable*

```
throw new Exception("Mensaje de error...");
```

Cuando se lanza una excepción:

- Se sale inmediatamente del bloque de código actual.
- Si el bloque tiene asociada una cláusula *catch* adecuada para el tipo de la excepción generada, se ejecuta el cuerpo de la cláusula *catch*.
- Si no, se sale inmediatamente del bloque (o método) dentro del cual está el bloque en el que se produjo la excepción y se busca una cláusula *catch* apropiada.
- El proceso continúa hasta llegar al método main de la aplicación. Si ahí tampoco existe una cláusula *catch* adecuada, la máquina virtual Java finaliza su ejecución con un mensaje de error.

Propagación de Excepciones (throws) Si en el cuerpo de un método se lanza una excepción -de un tipo derivado de la clase *Exception*-, en la cabecera del método hay que añadir la cláusula *throws* que incluye una lista de los tipos de excepciones que se pueden producir al invocar el método.

Ejemplo:

```
public String leerFichero (String nombreFichero)
throws IOException
...
```

Las excepciones de tipo *RuntimeException* (que son muy comunes) no es necesario declararlas en la cláusula throws. Al implementar un método, hay que decidir si las excepciones se propagarán hacia arriba (*throws*) o se capturarán en el propio método (*catch*)

1. Ejemplo de un método que propaga una excepción:

```
public void f() throws IOException {
  // Fragmento de codigo que puede
  // lanzar una excepción de tipo IOException
}
```

NOTA: Un método puede lanzar una excepción al crear explícitamente un objeto *Throwable* y lanzarlo con *throw*, o bien porque llame a un método que genere la excepción y no la capture.

2. Ejemplo de un método equivalente que no propaga la excepción:

```
public void f() {
  // Fragmento de código libre de excepciones
  try {
    // Fragmento de codigo que puede
    // lanzar una excepcion de tipo IOException
    // (p.ej. Acceso a un archivo)
  } catch (IOException error) {
    // Tratamiento de la excepcion
  } finally {
    // Liberar recursos (siempre se hace)
  }
}
```

Creación de nuevos tipos de excepciones Un nuevo tipo de excepción puede crearse fácilmente, basta con definir una subclase de un tipo de excepción ya existente.

```
public DivideByZeroException
extends ArithmeticException {
  public DivideByZeroException(String Message) {
    super(message);
  }
}
```

Una excepción de este tipo puede entonces lanzarse como cualquier otra excepción:

```
public double dividir(int num, int den)
throws DivideByZeroException {
  if (den==0) throw new DivideByZeroException("Error!");
  return ((double) num/(double)den);
}
```

NOTA: Las aplicaciones suelen definir sus propias subclases de la clase *Exception* para representar situaciones excepcionales específicas de cada aplicación.

Excepciones en Python Es posible escribir programas que gestionen determinadas excepciones. Véase el siguiente ejemplo, que le pide al usuario una entrada hasta que ingrese un entero válido, pero permite al usuario interrumpir el programa (usando Control-C o lo que soporte el sistema operativo); nótese que una interrupción generada por el usuario es señalizada generando la excepción KeyboardInterrupt.

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops! That was no valid number. Try again...")
```

La sentencia try funciona de la siguiente manera:

- Primero, se ejecuta la cláusula try (la(s) linea(s) entre las palabras reservadas try y la except).
- Si no ocurre ninguna excepción, la cláusula except se omite y la ejecución de la cláusula try finaliza.
- Si ocurre una excepción durante la ejecución de la cláusula try, se omite el resto de la cláusula. Luego, si su tipo coincide con la excepción nombrada después de la palabra clave except, se ejecuta la cláusula except, y luego la ejecución continúa después del bloque try/except.

- Si ocurre una excepción que no coincida con la excepción nombrada en la cláusula except, se pasa a las declaraciones try externas; si no se encuentra ningún controlador, de trata de una excepción no controlada y la ejecución se detiene con un mensaje de error.

Una declaración try puede tener más de una cláusula except, para especificar gestores para diferentes excepciones. Como máximo, se ejecutará un gestor. Los gestores solo manejan las excepciones que ocurren en la cláusula try correspondiente, no en otros gestores de la misma declaración try. Una cláusula except puede nombrar múltiples excepciones como una tupla entre paréntesis, por ejemplo:

```
except (RuntimeError, TypeError, NameError):
    pass
```

Una clase en una cláusula de excepción coincide con excepciones que son instancias de la clase misma o de una de sus clases derivadas (pero no al revés: una cláusula de excepción que enumera una clase derivada no coincide con instancias de sus clases base). Por ejemplo, el siguiente código imprimirá B, C, D en ese orden.

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass
for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Nótese que si las cláusulas except estuvieran invertidas (con except B primero), habría impreso B, B, B – se usa la primera cláusula except coincidente.

Cuando ocurre una excepción, puede tener un valor asociado, también conocido como el argumento de la excepción. La presencia y el tipo de argumento depende del tipo de excepción.

La cláusula except puede especificar una variable después del nombre de la excepción. La variable está ligada a la instancia de la excepción, que normalmente tiene un atributo args que almacena los argumentos. Por conveniencia, los tipos de excepción incorporados definen __str__() para imprimir todos los argumentos sin acceder explícitamente a .args.

```
try:
    raise Exception('spam', 'eggs')
except Exception as inst:
    print(type(inst)) # the exception type
    print(inst.args) # arguments stored in .args
    print(inst) # __str__ allows args to be printed directly,
    # but may be overridden in exception subclasses
    x, y = inst.args # unpack args
    print('x =', x)
    print('y =', y)
```

```
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

La salida __str__() de la excepción se imprime como la última parte ("detalle") del mensaje para las excepciones no gestionadas.

BaseException es la clase base común de todas las excepciones. Una de sus subclases, Exception, es la clase base de todas las excepciones no fatales. Las excepciones que no son subclases de Exception no se suelen manejar, porque se utilizan para indicar que el programa debe terminar. Entre ellas se incluyen SystemExit, que es lanzada por sys.exit() y KeyboardInterrupt, que se lanza cuando un usuario desea interrumpir el programa.

Exception se puede utilizar como un comodín que atrapa (casi) todo. Sin embargo, es una buena práctica ser lo más específico posible con los tipos de excepciones que pretendemos manejar, y permitir que cualquier excepción inesperada se propague.

El patrón más común para gestionar Exception es imprimir o registrar la excepción y luego volver a re-lanzarla (permitiendo a un llamador manejar la excepción también):

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

La declaración try ... except tiene una cláusula else opcional, que, cuando está presente, debe seguir todas las cláusulas except. Es útil para el código que debe ejecutarse si la cláusula try no lanza una excepción. Por ejemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

El uso de la cláusula else es mejor que agregar código adicional en la cláusula try porque evita capturar accidentalmente una excepción que no fue generada por el código que está protegido por la declaración try ... except.

Los gestores de excepciones no sólo gestionan excepciones que ocurren inmediatamente en la cláusula try, sino también aquellas que ocurren dentro

de funciones que son llamadas (incluso indirectamente) en la cláusula try. Por ejemplo:

```
def this_fails():
    x = 1/0
try:
    this_fails()
except ZeroDivisionError as err:
    print('Handling run-time error:', err)
```

```
Handling run-time error: division by zero
```

Lanzando excepciones La declaración raise permite al programador forzar a que ocurra una excepción específica. Por ejemplo:

```
raise NameError('HiThere')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

El único argumento de raise indica la excepción a lanzar. Debe ser una instancia de excepción o una clase de excepción (una clase que derive de BaseException, como Exception o una de sus subclases). Si se pasa una clase de excepción, se instanciará implícitamente llamando a su constructor sin argumentos:

```
raise ValueError # shorthand for 'raise ValueError()'
```

Si es necesario determinar si una excepción fue lanzada pero sin intención de gestionarla, una versión simplificada de la instrucción raise te permite relanzarla:

```
try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')
    raise
```

```
An exception flew by!
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
NameError: HiThere
```

Encadenamiento de excepciones Si se produce una excepción no gestionada dentro de una sección except, se le adjuntará la excepción que se está gestionando y se incluirá en el mensaje de error:

```
try:
    open("database.sqlite")
except OSError:
    raise RuntimeError("unable to handle error")
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: unable to handle error
```

Para indicar que una excepción es consecuencia directa de otra, la sentencia raise permite una cláusula opcional from:

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

Esto puede resultar útil cuando está transformando excepciones. Por ejemplo:

```
def func():
    raise ConnectionError
try:
    func()
except ConnectionError as exc:
    raise RuntimeError('Failed to open database') from exc
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

También permite deshabilitar el encadenamiento automático de excepciones utilizando el modismo from None:

```
try:
open('database.sqlite')
except OSError:
raise RuntimeError from None
```

```
Traceback (most recent call last):
File "<stdin>", line 4, in <module>
RuntimeError
```

Para obtener más información sobre la mecánica del encadenamiento, consulte Excepciones incorporadas.

Excepciones definidas por el usuario Los programas pueden nombrar sus propias excepciones creando una nueva clase excepción. Las excepciones, típicamente, deberán derivar de la clase Exception, directa o indirectamente. Las clases de Excepción pueden ser definidas de la misma forma que cualquier otra clase, pero es habitual mantenerlas lo más simples posible, a menudo ofreciendo solo un número de atributos con información sobre el error que leerán los gestores de la excepción.

La mayoría de las excepciones se definen con nombres acabados en «Error», de manera similar a la nomenclatura de las excepciones estándar. Muchos módulos estándar definen sus propias excepciones para reportar errores que pueden ocurrir en funciones propias.

Definiendo acciones de limpieza La declaración try tiene otra cláusula opcional cuyo propósito es definir acciones de limpieza que serán ejecutadas bajo ciertas circunstancias. Por ejemplo:

```
try:
    raise KeyboardInterrupt
finally:
    print('Goodbye, world!')
```

```
Goodbye, world!
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

Si una cláusula finally está presente, el bloque finally se ejecutará al final antes de que todo el bloque try se complete. La cláusula finally se ejecuta independientemente de que la cláusula try produzca o no una excepción. Los siguientes puntos explican casos más complejos en los que se produce una excepción:

- Si ocurre una excepción durante la ejecución de la cláusula try, la excepción podría ser gestionada por una cláusula except. Si la excepción no es gestionada por una cláusula except, la excepción es relanzada después de que se ejecute el bloque de la cláusula finally.

- Podría aparecer una excepción durante la ejecución de una cláusula except o else. De nuevo, la excepción será relanzada después de que el bloque de la cláusula finally se ejecute.
- Si la cláusula finally ejecuta una declaración break, continue o return, las excepciones no se vuelven a lanzar.
- Si el bloque try llega a una sentencia break, continue o return, la cláusula finally se ejecutará justo antes de la ejecución de dicha sentencia.
- Si una cláusula finally incluye una sentencia return, el valor retornado será el de la cláusula finally, no la del de la sentencia return de la cláusula try.

Por ejemplo:

```
def bool_return():
    try:
        return True
    finally:
        return False
bool_return()
```

```
False
```

Un ejemplo más complicado:

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
    print("executing finally clause")
divide(2, 1)
```

```
result is 2.0
executing finally clause
    divide(2, 0)
division by zero!
executing finally clause
    divide("2", "1")
executing finally clause
Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
     File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Como se puede ver, la cláusula finally siempre se ejecuta. La excepción TypeError lanzada al dividir dos cadenas de texto no es gestionado por la cláusula except y por lo tanto es relanzada luego de que se ejecuta la cláusula finally.

En aplicaciones reales, la cláusula finally es útil para liberar recursos externos (como archivos o conexiones de red), sin importar si el uso del recurso fue exitoso.

Acciones predefinidas de limpieza algunos objetos definen acciones de limpieza estándar para llevar a cabo cuando el objeto ya no necesario, independientemente de que las operaciones sobre el objeto hayan sido exitosas o no. Véase el siguiente ejemplo, que intenta abrir un archivo e imprimir su contenido en la pantalla:

```
for line in open("myfile.txt"):
    print(line, end="")
```

El problema con este código es que deja el archivo abierto por un periodo de tiempo indeterminado luego de que esta parte termine de ejecutarse. Esto no es un problema en scripts simples, pero puede ser un problema en aplicaciones más grandes. La declaración with permite que los objetos como archivos sean usados de una forma que asegure que siempre se los libera rápido y en forma correcta:

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

Una vez que la declaración se ejecuta, el fichero f siempre se cierra, incluso si aparece algún error durante el procesado de las líneas. Los objetos que, como los ficheros, posean acciones predefinidas de limpieza lo indicarán en su documentación.

Lanzando y gestionando múltiples excepciones no relacionadas Hay situaciones en las que es necesario informar de varias excepciones que se han producido. Este es a menudo el caso en los marcos de concurrencia, cuando varias tareas pueden haber fallado en paralelo, pero también hay otros casos de uso en los que es deseable continuar la ejecución y recoger múltiples errores en lugar de lanzar la primera excepción.

El incorporado ExceptionGroup envuelve una lista de instancias de excepción para que puedan ser lanzadas juntas. Es una excepción en sí misma, por lo que puede capturarse como cualquier otra excepción:

```
def f():
    excs = [OSError('error 1'), SystemError('error 2')]
    raise ExceptionGroup('there were problems', excs)
f()
```

```
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| File "<stdin>", line 3, in f
| ExceptionGroup: there were problems
+-+---------- 1 ----------
| OSError: error 1
+---------- 2 ----------
| SystemError: error 2
+------------------------
```

```
try:
    f()
except Exception as e:
    print(f'caught {type(e)}: e')
```

```
caught <class 'ExceptionGroup'>: e
```

Utilizando except* en lugar de except, podemos manejar selectivamente sólo las excepciones del grupo que coincidan con un determinado tipo. En el siguiente ejemplo, que muestra un grupo de excepciones anidado, cada cláusula except* extrae del grupo las excepciones de un tipo determinado, mientras que deja que el resto de excepciones se propaguen a otras cláusulas y, finalmente, se vuelvan a lanzar:

```
def f():
    raise ExceptionGroup(
        "group1",
        [
            OSError(1),
            SystemError(2),
            ExceptionGroup(
                "group2",
                [
                    OSError(3),
                    RecursionError(4)
                ]
            )
        ]
    )
try:
    f()
except* OSError as e:
    print("There were OSErrors")
except* SystemError as e:
    print("There were SystemErrors")
```

```
There were OSErrors
There were SystemErrors
```

```
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 2, in <module>
| File "<stdin>", line 2, in f
| ExceptionGroup: group1
+-+---------- 1 ----------
| ExceptionGroup: group2
+-+---------- 1 ----------
| RecursionError: 4
+------------------------
```

Tenga en cuenta que las excepciones anidadas en un grupo de excepciones deben ser instancias, no tipos. Esto se debe a que en la práctica las excepciones serían típicamente las que ya han sido planteadas y capturadas por el programa, siguiendo el siguiente patrón:

```
excs = []
for test in tests:
    try:
        test.run()
    except Exception as e:
    excs.append(e)
if excs:
    raise ExceptionGroup("Test Failures", excs)
```

Enriqueciendo excepciones con notas Cuando se crea una excepción para ser lanzada, normalmente se inicializa con información que describe el error que se ha producido. Hay casos en los que es útil añadir información después de que la excepción haya sido capturada. Para este propósito, las excepciones tienen un método add_note(note) que acepta una cadena y la añade a la lista de notas de la excepción. La representación estándar del rastreo incluye todas las notas, en el orden en que fueron añadidas, después de la excepción:

```
try:
    raise TypeError('bad type')
```

```
except Exception as e:
        e.add_note('Add some information')
        e.add_note('Add some more information')
        raise
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: bad type
Add some information
Add some more information
```

Por ejemplo, al recopilar excepciones en un grupo de excepciones, es posible que queramos añadir información de contexto para los errores individuales. A continuación, cada excepción del grupo tiene una nota que indica cuándo se ha producido ese error.

```
def f():
    raise OSError('operation failed')
excs = []
for i in range(3):
    try:
        f()
    except Exception as e:
        e.add_note(f'Happened in Iteration {i+1}')
        excs.append(e)
raise ExceptionGroup('We have some problems', excs)
```

```
+ Exception Group Traceback (most recent call last):
| File "<stdin>", line 1, in <module>
| ExceptionGroup: We have some problems (3 sub-exceptions)
+-+---------- 1 ----------
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| File "<stdin>", line 2, in f
| OSError: operation failed
```

```
| Happened in Iteration 1
+------------- 2 -------------
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| File "<stdin>", line 2, in f
| OSError: operation failed
| Happened in Iteration 2
+------------- 3 -------------
| Traceback (most recent call last):
| File "<stdin>", line 3, in <module>
| File "<stdin>", line 2, in f
| OSError: operation failed
| Happened in Iteration 3
+------------------------------
```

3.5 Altas, Bajas y Cambios

En programación, crear, leer, actualizar y borrar (con el acrónimo CRUD[31]) son las cuatro funciones básicas de la persistencia de Bases de Datos. Términos alternativos son usados a veces cuando se definen las cuatro funciones básicas de CRUD, como "recuperar" en vez de "leer", "modificar" en vez de "actualizar" o "destruir" en vez de "borrar". CRUD se usa también a veces para describir convenciones de interfaz de usuario que facilita la vista, búsqueda y modificación de la información; a menudo se usa en programación de formularios (Forms) e informes (Reports). El acrónimo puede extenderse a

[31]En informática, CRUD es el acrónimo de "Crear, Leer, Actualizar y Borrar" (del original en inglés: Create, Read, Update and Delete), que se usa para referirse a las funciones básicas en bases de datos o la capa de persistencia en un Software.

En algunos lugares, se utilizan las siglas ABM para lo mismo ("Alta, Baja y Modificación"), obviando la operación de "obtener"; el acrónimo ABC para "Altas, Bajas y Cambios"; ABML siendo la última letra (L) de "listar, listado o lectura"; ABMC siendo la 'C' de "Consulta"; o bien CLAB que sería la traducción literal del acrónimo ("Crear, Leer, Actualizar y Borrar") también se llega a usar el acrónimo ABCC ("Altas, Bajas, Cambios y Consultas").

CRUDL[32] para cubrir el listado de gran cantidad de datos que conllevan una complejidad tal como paginación cuando los registros de datos son demasiado grandes para alojarse fácilmente en memoria.

3.5.1 Ejemplo en Java

En esta sección, desarrollaremos un sistema a modo de ejemplo de Altas, Bajas y Cambios (ABC) además de Leer y Grabar los datos capturados -para manipular la lista de los registros usaremos un *ArrayList*[33]- usando programación genérica para desarrollar un par de ejemplos -Directorio Telefónico y Catálogo de CDs- que pueden ser usados por separado o como además se mostrará, en un arreglo de objetos que permite su manipulación de forma simultáneo, mediante un único sistema de cómputo que aproveche las facilidades de la programación genérica mostrada en la siguiente jerarquía de clases.

Para ello, primero desarrollamos una clase abstracta pura que defina los comportamientos que harán la manipulación de cada componente (registro) de nuestro sistema de Altas, Bajas y Cambios, mediante el siguiente código:

```
import java.io.Serializable;[34]
/// Clase Base para manipular registros
public abstract class Registro implements Serializable {
   private static final long serialVersionUID = 1L;
   abstract int visualizaRegistro();
   abstract int modificarRegistro();
   abstract int adicionaRegistro();
};
```

[32] El término fue popularizado por primera vez por James Martín en su libro del año 1980 Managing the Data-base Enviroment.

[33] Para manipular estructuras de datos que nos permita añadir, eliminar y modificar elementos —pueden ser objetos o elementos atómicos— de forma transparente para el programador en Java se desarrollo entre otras opciones al *ArrayList*. Esta clase permite almacenar datos en memoria de forma similar a los *Arrays*, con la ventaja de que el número de elementos que almacena, lo hace de forma dinámica.

[34] Para usar *ArrayList* y que nos permite leer y grabar al conjunto de registros de forma unificada, necesitamos usar *java.io.Serializable*. E implementar la clase *Registro* a partir de la clase *Serializable*, además de definir:
private static final long serialVersionUID = 1L;

Para luego definir los comportamientos mediante la herencia, para el directorio telefónico:

```
import java.util.Scanner;[35]
/// Clase para manipular registros telefonicos
  public class RegTelefonico extends Registro {
  private String Nombre;
  private String Direccion;
  private String Telefonos;
  /// Visualiza el contenido del registro
  int visualizaRegistro() {
    System.out.println("Nombre: " + Nombre);
    System.out.println("Direccion: " + Direccion);
    System.out.println("Telefonos: " + Telefonos);
    return 0; //Ok
  }
  /// Adiciona un registro
  int adicionaRegistro() {
    Scanner teclado = new Scanner(System.in);
    System.out.print("Nombre: ");
    Nombre = teclado.nextLine();
    System.out.print("Direccion: ");
    Direccion = teclado.nextLine();
    System.out.print("Telefonos: ");
    Telefonos = teclado.nextLine();
    return 0; // Ok
  }
  /// Modifica el contenido del registro
  int modificarRegistro() {
    Scanner teclado = new Scanner(System.in);
    // Visualiza el nombre y permite su modificacion
    System.out.println("Nombre: " + Nombre);
    System.out.print("Nuevo Nombre: ");
    Nombre = teclado.nextLine();
    System.out.println("Direccion: " + Direccion);
```

[35] En nuestro ejemplo, usaremos la clase *Scanner*, para solicitar en consola la captura de los datos en modo texto.

```
        System.out.print("Nueva Direccion: ");
        Direccion = teclado.nextLine();
        System.out.println("Telefonos: " + Telefonos);
        System.out.print("Nuevos Telefonos: ");
        Telefonos = teclado.nextLine();
        return 0; // Ok
    }
};
```

y del catálogo de CDs, mediante:

```
import java.util.Scanner;[36]
/// Clase para manipular registros de Cds
public class RegCDs extends Registro {
    private String Titulo;
    private String Artista;
    private int NumCanciones;
    private String Canciones;
    /// Visualiza el contenido del registro
    public int visualizaRegistro() {
        System.out.println("Titulo: " + Titulo);
        System.out.println("Artista: " + Artista);
        System.out.println("Numero de caciones: "
+ NumCanciones);
        System.out.println("Canciones: " + Canciones);
        return 0; //Ok
    }
    /// Adiciona un registro
    public int adicionaRegistro() {
        Scanner teclado = new Scanner(System.in);
        System.out.print("Titulo: ");
        Titulo = teclado.nextLine();
        System.out.print("Artista: ");
        Artista = teclado.nextLine();
        System.out.print("Numero de caciones: ");
        NumCanciones = teclado.nextInt();
```

[36] En nuestro ejemplo, usaremos la clase *Scanner*, para solicitar en consola la captura de los datos en modo texto.

```
        System.out.print("Canciones: ");
        Canciones = teclado.nextLine();
        Canciones = teclado.nextLine();
        return 0; // Ok
    }
    /// Modifica el contenido del registro
    int modificarRegistro() {
        Scanner teclado = new Scanner(System.in);
        // Visualiza el nombre y permite su modificacion
        System.out.println("Titulo: " + Titulo);
        System.out.print("Nuevo Titulo: ");
        Titulo = teclado.nextLine();
        System.out.println("Artista: " + Artista);
        System.out.print("Nuevo Artista: ");
        Artista = teclado.nextLine();
        System.out.println("Numero de caciones: "
+ NumCanciones);
        System.out.print("Nuevo Numero de caciones: ");
        NumCanciones = teclado.nextInt();
        System.out.println("Canciones: " + Canciones);
        System.out.print("Nuevas Canciones: ");
        Canciones = teclado.nextLine();
        Canciones = teclado.nextLine();
        return 0; // Ok
    }
};
```

Los códigos anteriores definen los comportamientos genéricos que son la implementación de los comportamientos abstractos:

```
abstract int visualizaRegistro();
abstract int modificarRegistro();
abstract int adicionaRegistro();
```

Ahora implementamos la clase abstracta que soportará los comportamientos que manipulen a las clases anteriores mediante un *ArrayList* y que permitan la implementación de las Altas, Bajas y Cambios, mediante:

```
import java.io.*;[37]
import java.util.*;[38]
/// Clase base para manipular Altas-Bajas-Cambios.
public abstract class EstructuraABC {
  /// Lista
  public ArrayList<Registro> lista;
  public EstructuraABC() {
    lista = new ArrayList<Registro>();
  }
  /// Borra todo el contenido de la lista
  private void borrarTodo() {
    lista.clear();
  }
  /// Adiciona un registro al final de la lista
  abstract int adicionar();
  /// Visualiza el contenido del registro solicitado
  public int visualiza(int pos) {
    if (pos >= 0 && pos < lista.size()) {
      lista.get(pos).visualizaRegistro();
      return 0;
    }
    return 1;
  }
  /// Visualiza todo el contenido de la lista
  public int visualizaTodos() {
    Iterator<Registro> iter = lista.iterator();
    while (iter.hasNext()) {
      iter.next().visualizaRegistro();
    }
    return 0;
  }
  /// Visualiza el registro de la lista solicitado
  int borrar(int pos) {
    if (pos >= 0 && pos < lista.size()) {
      lista.remove(pos);
```

[37] Para permitir leer y grabar archivos.
[38] Para definir *ArrayList*.

```
            return 0;
        }
        return 1;
    }
    /// Modifica el registro de la lista solicitado
    int modificar(int pos) {
        if (pos >= 0 && pos < lista.size()) {
            lista.get(pos).modificarRegistro();
            return 0;
        }
        return 1;
    }
    /// Leer una lista de disco
    @SuppressWarnings("unchecked")
    int leer(String arch) {
        try {
            FileInputStream fileIn = new FileInputStream(arch);
            ObjectInputStream in = new ObjectInputStream(fileIn);
            Object obj = in.readObject();
            lista = (ArrayList<Registro>) obj;
            in.close();
            fileIn.close();
            visualizaTodos();
        } catch(Exception e){e.printStackTrace();}
        return 0;
    }
    /// Grabar una lista en disco
    int grabar(String arch) {
        try {
            FileOutputStream fileOut = new FileOutputStream(arch);
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(lista);
            out.close();
            fileOut.close();
        } catch(Exception e){}
        return 0;
    }
    /// Regresa el numero de registros en la lista
```

```
    int regresaNRegs() {
      return lista.size();
    }
};
```

Una vez definidos los códigos anteriores, podemos ya implementar la clase que permita definir mediante herencia la manipulación de los registros telefónicos haciendo uso de la herencia del código de Altas, Bajas y Cambios, mediante:

```
/// Clase para manipular directorio telefonicos
class DirecTelefonico extends EstructuraABC {
  // Adiciona registro al final de la lista
  public int adicionar() {
    RegTelefonico reg = new RegTelefonico();
    reg.adicionaRegistro();
    lista.add(reg);
    return 0; // Ok
  }
};
```

Y para el Catálogo de CDs, mediante:

```
/// Clase para manipular catalogos de CDs
public class CatalogoCDs extends EstructuraABC {
  // Adiciona registro al final de la lista
  int adicionar() {
    RegCDs reg = new RegCDs();
    reg.adicionaRegistro();
    lista.add(reg);
    return 0;
  }
};
```

Ahora definiremos un menú para que el usuario manipule nuestros sistemas de Altas, Bajas y Cambios, por ejemplo para el Directorio Telefónico, mediante:

```
import java.util.Scanner;[39]
public class Test {
  public static void limpiar() {
    for (int i = 0; i < 20; i++) System.out.println("");
  }
  public static void visualizaMenu() {
    System.out.println("Menu");
    System.out.println("");
    System.out.println("1) Agregar");
    System.out.println("2) Modificar");
    System.out.println("3) Borrar");
    System.out.println("4) Visualizar todos");
    System.out.println("8) Leer");
    System.out.println("9) Grabar");
    System.out.println("0) Salir");
  }
  // Funcion Principal ....
  public static void main(String[] args) {
    Scanner teclado = new Scanner(System.in);
    // Manipulacion del directorio telefonico
    DirecTelefonico man = new DirecTelefonico();
    limpiar();
    int op = 1, reg;
    do {
      System.out.println("Numero de registros: "
+ man.regresaNRegs());
      visualizaManu();
      System.out.println("");
      System.out.println("Opcion: ");
      op = teclado.nextInt();
      switch(op) {
        case 1:
          man.adicionar();
          break;
        case 2:
```

[39] En nuestro ejemplo, usaremos la clase *Scanner*, para solicitar en consola la captura de los datos en modo texto.

```
                System.out.println("Registro: ");
                reg = teclado.nextInt();
                man.modificar(reg);
                break;
            case 3:
                System.out.println("Registro: ");
                reg = teclado.nextInt();
                man.borrar(reg);
                break;
            case 4:
                man.visualizaTodos();
                break;
            case 8:
                man.leer("DirTelefonico.dat");
                break;
            case 9:
                man.grabar("DirTelefonico.dat");
                break;
          }
        } while(op != 0);
    }
}
```

Y para el Catálogo de CDs, mediante:

```
import java.util.Scanner;[40]
public class Test {
    public static void limpiar() {
        for (int i = 0; i < 20; i++) System.out.println("");
    }
    public static void visualizaMenu() {
        System.out.println("Menu");
        System.out.println("");
        System.out.println("1) Agregar");
        System.out.println("2) Modificar");
        System.out.println("3) Borrar");
```

[40] En nuestro ejemplo, usaremos la clase *Scanner*, para solicitar en consola la captura de los datos en modo texto.

```
        System.out.println("4) Visualizar todos");
        System.out.println("8) Leer");
        System.out.println("9) Grabar");
        System.out.println("0) Salir");
    }
    // Funcion Principal ....
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        // Manipulacion del directorio telefonico
        CatalogoCDs man = new CatalogoCDs();
        limpiar();
        int op = 1, reg;
        do {
            System.out.println("Numero de registros: "
+ man.regresaNRegs());
            visualizaManu();
            System.out.println("");
            System.out.println("Opcion: ");
            op = teclado.nextInt();
            switch(op) {
                case 1:
                    man.adicionar();
                    break;
                case 2:
                    System.out.println("Registro: ");
                    reg = teclado.nextInt();
                    man.modificar(reg);
                    break;
                case 3:
                    System.out.println("Registro: ");
                    reg = teclado.nextInt();
                    man.borrar(reg);
                    break;
                case 4:
                    man.visualizaTodos();
                    break;
                case 8:
                    man.leer("DirTelefonico.dat");
```

```
                    break;
                case 9:
                    man.grabar("DirTelefonico.dat");
                    break;
            }
        } while(op != 0);
    }
}
```

Aún mejor, podemos usar el Directorio telefónico y el Catálogo de CDs simultáneamente, permitiendo al usuario cambiarse entre uno u otro según sea necesario, mediante:

```
import java.util.Scanner;[41]
public class Test {
    public static String[] Archivo = {"DirTelefonico.dat", "Cat-
alogoCDs.dat"};
    public static void limpiar() {
        for (int i = 0; i < 20; i++) System.out.println("");
    }
    public static void visualizaMenu() {
        System.out.println("Menu");
        System.out.println("");
        System.out.println("1) Agregar");
        System.out.println("2) Modificar");
        System.out.println("3) Borrar");
        System.out.println("4) Visualizar todos");
        System.out.println("8) Leer");
        System.out.println("9) Grabar");
        System.out.println("\n10) Cambiar entre DirecTelefon-
ico y CatalogoCDs");
        System.out.println("0) Salir");
    }
    // Funcion Principal ....
    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
```

[41] En nuestro ejemplo, usaremos la clase *Scanner*, para solicitar en consola la captura de los datos en modo texto.

```
        EstructuraABC []man= new EstructuraABC[2];
        man[0] = new DirecTelefonico();
        man[1] = new CatalogoCDs();
        limpiar();
        int op = 1, act = 0, reg;
        do {
          System.out.println("Numero de registros: "
+ man[act].regresaNRegs() + " en " + Archivo[act]);
          System.out.println("");
          visualizaMenu();
          System.out.println("Opcion: ");
          op = teclado.nextInt();
          switch(op) {
            case 1:
              man[act].adicionar();
              break;
            case 2:
              System.out.println("Registro: ");
              reg = teclado.nextInt();
              man[act].modificar(reg);
              break;
            case 3:
              System.out.println("Registro: ");
              reg = teclado.nextInt();
              man[act].borrar(reg);
              break;
            case 4:
              man[act].visualizaTodos();
              break;
            case 8:
              man[act].leer(Archivo[act]);
              break;
            case 9:
              man[act].grabar(Archivo[act]);
              break;
            case 10:
              if (act == 0) act = 1;
              else act = 0;
```

```
                break;
              default:
                System.out.println("Opcion no reconocidad");
            }
          } while(op != 0);
        }
      }
```

De esta forma, tenemos una clase genérica de Altas, Bajas y Cambios que nos permite manipular uno o múltiples ejemplos -Directorio Telefónico y el Catálogo de CDs- para hacer Altas, Bajas y Cambios en un solo sistema computacional que es flexible, extendible y de fácil mantenimiento, además de permitir el manejo simultáneo de múltiples sistemas de ABC.

3.5.2 Ejemplo en Python

En esta sección, desarrollaremos un sistema a modo de ejemplo de Altas, Bajas y Cambios (ABC) además de Leer y Grabar los datos capturados -para manipular la lista de los registros usaremos *list*[42] *y pickle*[43]- usando programación genérica para desarrollar un par de ejemplos de productos -Televisor y Reproductor- que pueden ser usados por separado o como además se mostrará, en una lista de objetos que permite su manipulación de forma simultánea, mediante un único sistema de cómputo que aproveche las facilidades de la programación mostrada en la siguiente jerarquía de clases.

Para ello, primero desarrollamos una clase base *Producto* que defina los comportamientos comunes de nuestro sistema de Altas, Bajas y Cambios, mediante el siguiente código:

```
class Producto:
  """Definicion de la clase producto"""
  def __init__(self):
    """Constructor de la clase producto"""
    self.__id = None
```

[42] Para manipular estructuras de datos que nos permita añadir, eliminar y modificar elementos -pueden ser objetos o elementos atómicos- de forma transparente para el programador. Esta clase permite almacenar datos en memoria, con la ventaja de que el número de elementos que almacena lo hace de forma dinámica.

[43] Alternativamente mostramos el mismo ejemplo usando shelve en vez de pickle en la siguiente jerarquía de clases.

```
        self.__nomb = None
        self.__precio = None
        self.__cant = None
    def inicializa(self, id, nomb, precio, cant):
        """Inicializa la clase producto"""
        self.__id = id
        self.__nomb = nomb
        self.__precio = precio
        self.__cant = cant
    def captura(self):
        """Captura la clase producto"""
        self.__id = input("Identificador: ")
        self.__nomb = input("Nombre: ")
        self.__precio = float(input("Precio: "))
        self.__cant = int(input("Cantidad: "))
    def identificador(self):
        """Retorna el identificador producto"""
        return self.__id
    # Acceso al valor self.__id mediante la variable id
    id = property(identificador)
    def visualizar(self):
        """Visualiza la clase producto"""
        print(self)
    def __repr__(self):
        """Visualiza la clase producto"""
        a = "\nIdentificador: {0}\n".format(self.__id)
        a += "Nombre: {0}\n".format(self.__nomb)
        a += "Precio: {:,.2f}\n".format(self.__precio)
        a += "Cantidad: {0}\n".format(self.__cant)
        return a
```

Para luego definir los comportamientos mediante la herencia, para el producto Televisor:

```
class Televisor(Producto):
    """Definicion de la clase Televisor"""
    def __init__(self):
        """Constructor de la clase Televisor"""
```

```
        Producto.__init__(self)
        self.__pulg = None
        self.__resol = None
    def inicializa(self, id, nomb, precio, cant, pulg, resol):
        """Inicializa la clase Televisor"""
        Producto.inicializa(self, id, nomb, precio, cant)
        self.__pulg = pulg
        self.__resol = resol
    def captura(self):
        """Captura la clase Televisor"""
        Producto.captura(self)
        self.__pulg = int(input("Pulgadas: "))
        self.__resol = input("Resolucion: ")
    def __repr__(self):
        """Visualiza la clase Televisor"""
        a = Producto.__repr__(self)
        a += "Pulgadas: {0}\n".format(self.__pulg)
        a += "Resolucion: {0}\n".format(self.__resol)
        return a
```

y el producto Reproductor, mediante:

```
class Reproductor(Producto):
    """Definicion de la clase Reproductor"""
    def __init__(self):
        """Constructor de la clase Reproductor"""
        Producto.__init__(self)
        self.__bocinas = None
    def inicializa(self, id, nomb, precio, cant, bocinas):
        """Inicializa la clase Reproductor"""
        Producto.inicializa(self, id, nomb, precio, cant)
        self.__bocinas = bocinas
    def captura(self):
        """Captura la clase Reproductor"""
        Producto.captura(self)
        self.__bocinas = int(input("Bocinas: "))
    def __repr__(self):
        """Visualiza la clase Reproductor"""
```

```
        a = Producto.__repr__(self)
        a += "Bocinas: {0}\n".format(self.__bocinas)
        return a
```

Ahora implementamos la clase que soportará los comportamientos que manipulen a las clases anteriores mediante una lista *list* y que permitan la implementación de las Altas, Bajas y Cambios, mediante:

```
class Prueba:
    """ Clase de prueba """
    def __init__(self):
        """Constructor de la clase"""
        self.__lista = [] # Lista para los objetos creados
        self.__archivo = "ArchivoBD"
        self.__msw = 1
        if os.path.exists(self.__archivo):
            with open(self.__archivo, "rb") as fp:
                self.__lista = pickle.load(fp)
    def __del__(self):
        """Destructor"""
    def menu(self):
        """menu """
        sw = {
            "1": self.altas,
            "2": self.bajas,
            "3": self.cambios,
            "4": self.visualiza,
            "9": self.salir,
        }
        while self.__msw:
            # os.system('clear') # NOTA para windows tienes que
cambiar clear por cls
            print("Selecciona una opcion")
            print("\t1 - Altas")
            print("\t2 - Bajas")
            print("\t3 - Cambios")
            print("\t4 - Visualizar")
            print("\t9 - salir")
```

```
            # solicituamos una opcion al usuario
            opcionMenu = input("inserta un numero >> ")
            sw.get(opcionMenu, self.otro)()
        def altas(self):
            """ Altas """
            print("")
            opcion = input("Deseas dar de alta un Televisor (1) o un
    reproductor(2)? ")
            if opcion == "1":
                a = Televisor()
                a.captura()
                for i in self.__lista:
                    if a.id == i.id:
                        print("Error identificador existe")
                        break
                    else:
                        self.__lista.append(a)
            elif opcion == "2":
                    b = Reproductor()
                    b.captura()
                    for i in self.__lista:
                        if b.id == i.id:
                            print("Error identificador existe")
                            break
                        else:
                            self.__lista.append(b)
        def bajas(self):
            """ Bajas"""
            print("")
            id = input("Identificador a borrar?")
            for i in self.__lista:
                if id == i.id:
                    print("identificador borrado")
                    print(i.identificador())
                    self.__lista.remove(i)
        def cambios(self):
            """ Cambios"""
            print("")
```

```
            id = input("Identificador a cambiar?")
            for i in self.__lista:
                if id == i.id:
                    i.captura()
                else:
                    print("identificador no encontrado")
        def visualiza(self):
            """ Visualizar"""
            # print("Visualiza la lista de articulos desde la lista")
            # print(self.__lista) # Visualiza el objeto lista
            print("Visualiza la lista de articulos")
            for i in self.__lista:
                # i.visualizar()
                # i.visualizaAtributosJeraquia()
                print(i)
        def salir(self):
            """Salir"""
            self.__msw = 0
            with open(self.__archivo, "wb") as fp:
                pickle.dump(self.__lista, fp)
        def otro(self):
            """En caso de no ser alguna de las opciones definidas"""
            print("")
            input(
            "No has pulsado ninguna opcion correcta...\npulsa una
    tecla para continuar"
            )
```

Una vez definidos los códigos anteriores, podemos probar el programa mediante:

```
"""
Prueba de las clases
"""
if __name__ == "__main__":
    a = Prueba()
    a.menu()
```

De esta forma, tenemos una clase de Altas, Bajas y Cambios que nos permite manipular uno o múltiples ejemplos -Televisor y Reproductor- en un solo sistema computacional que es flexible, extendible y de fácil mantenimiento, además de permitir el manejo simultáneo de múltiples sistemas de ABC.