



Universidad Nacional Autónoma de México
Facultad de Ingeniería
Programación orientada a objetos
Tema 5:
MANEJO DE EXCEPCIONES Y ERRORES



5 Manejo de excepciones y errores

Objetivo: Clasificar los diferentes tipos de errores y excepciones para generar programas y aplicaciones con calidad.



5 Manejo de excepciones y errores

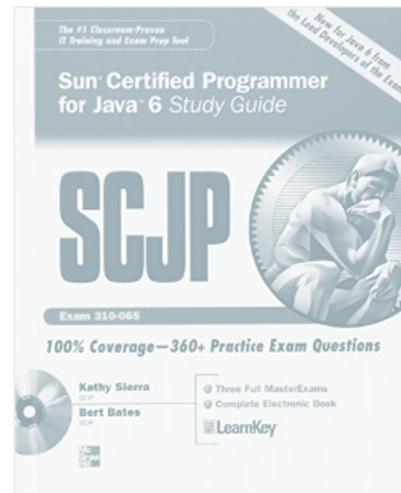
5.1 Definición y diferencia entre error y excepción.

5.2 Jerarquía de clases de errores.

5.3 Estructura try-catch-finally.

5.4 Manejo de errores y excepciones.

Bibliografía



**Sierra Katy, Bates Bert
SCJP Sun Certified Programmer for Java 6 Study Guide
Mc Graw Hill**

Bibliografía



*Carlos Blé Jurado
Diseño Ágil con TDD,
2010*



**“Theory is when you know something, but it doesn’t work.
Practice is when something works, but you don’t know why.
Programmers combine theory and practice: Nothing works and
they don’t know why.”**

**Unknown
@CodeWisdom**

PRUEBAS DE SOFTWARE



Las aplicaciones son propensas a presentar errores debido a diferentes factores: los tiempos de desarrollo, desarrolladores inexpertos, entornos de programación, las diferencias entre versiones, etc.

Por tanto, es necesario asegurar en la medida de lo posible la calidad del producto de software (QA).



El objetivo principal de las pruebas de software debe ser verificar que se cumplan con las especificaciones planteadas en los requerimientos iniciales, así como eliminar los errores que se hayan cometido durante el desarrollo de la aplicación.

La etapa de pruebas es crítica dentro del ciclo de vida del software, por lo que el proceso de testing debe seguir metodologías y métricas generales que revisen los aspectos fundamentales del sistema.



Las pruebas de software proporcionan información objetiva sobre la calidad del producto. Las pruebas son actividades que se encuentran dentro del desarrollo de software.

Es importante que la persona que desarrolla no sea la misma que la que prueba, debido a que normalmente nadie quiere sabotear su trabajo y, por ende, los errores que encuentra el desarrollador son siempre menores a los que puede encontrar un tester.



Las pruebas de software se clasifican en diferentes niveles dependiendo de la etapa donde se realicen las mismas, las cuales son:

- **Pruebas unitarias**
- **Pruebas de integración**
- **Pruebas de sistema**
- **Pruebas de aceptación**

A su vez, en cada nivel se podrían ejecutar pruebas funcionales, no funcionales, de arquitectura o asociadas al cambio de productos.

Pruebas unitarias

Las pruebas unitarias o de componente consisten en ejecutar actividades que permitan verificar que los componentes unitarios están codificados de manera robusta, esto es, que controlan el ingreso de datos erróneos o inesperados, demostrando que se tiene la capacidad de controlar errores de manera adecuada.

Es importante que la funcionalidad de cada componente unitario sea cubierta, por al menos, dos casos de prueba, probando, al menos, una funcionalidad positiva (happy path) y una negativa.

Pruebas de integración

Estas pruebas consisten en la comprobación de que elementos del software que interactúan entre sí funcionan de manera correcta con base en los requerimientos del usuario.

Tanto las pruebas unitarias como las pruebas de integración son ejecutadas por el proveedor (desarrollador) directamente, preferentemente por un equipo interno de pruebas.

Pruebas de sistema

Las pruebas de sistema deben realizarse por un tercero. Consiste en verificar que la funcionalidad total del sistema haya sido implementada por el proveedor con base en los requerimientos iniciales del cliente.

Los casos de prueba deben incluir requerimientos funcionales y no funcionales. Este tipo de pruebas se debe realizar en el ambiente de calidad, el cual debe ser similar al ambiente de producción.

Pruebas de aceptación

Las pruebas de aceptación las debe realizar el cliente, aquí es recomendable que las personas que realizan estas pruebas sean diferentes a las que fueron guiando el proceso de desarrollo con el proveedor.

Estas pruebas se realizan en ambientes productivos y son validadas por el usuario final del sistema.

Principios FIRST

Estos principios son utilizados para escribir buenas pruebas unitarias (Unit Test). Normalmente, cuando se desarrolla una aplicación lo menos esperado es tener problemas desarrollando las pruebas unitarias, los principios FIRST pueden ayudar a hacer más fácil el desarrollo de las mismas.



Los principios FIRST para escribir pruebas unitarias es el acrónimo de:

- **Fast (Rápido)**
- **Isolated (Aislado)**
- **Repeatable (Repetible)**
- **Self-validating (Validación propia)**
- **Timely (En tiempo)**

Siguiendo estos 5 principios se pueden obtener pruebas unitarias más robustas y, por ende, que el código de la aplicación sea más estable.

Fast (Rápido)

Las pruebas unitarias deben ser rápidas, de otra manera las pruebas atrasarán las etapas de desarrollo, dando tiempos más largos para pasar la prueba o para fallar.

Normalmente, en un sistema suficientemente largo, deben existir unas 2000 pruebas unitarias. En promedio, una prueba unitaria toma 200 milisegundos para ejecutarse, tomando 6.5 minutos para ejecutar todas las pruebas.



A pesar de que 6.5 minutos no se percibe como muy tardado, hay que considerar que las pruebas se pueden correr varias veces al día (por las correcciones que puedan surgir en cada revisión).

Además, si se agregan funcionalidades al sistema, las pruebas van a crecer, lo que incrementaría también el tiempo de ejecución.



Una de las causas principales que hacen que una prueba sea lenta es su dependencia con males necesarios como una base de datos, consulta de archivos, conexiones de red, etc.

Los grados de dependencia se pueden manejar a través de una prueba Mock. Una prueba Mock es aquella que utiliza objetos simulados para que la prueba sea suficientemente realista y rápida.



Por ejemplo, se tiene una prueba que consiste en validar el comportamiento un método m. Dentro de m se necesita un DAO (Data Access Object) para obtener información de un archivo.

Sin embargo, para la prueba no es necesario realizar la conexión puesto que lo que se está validando es el funcionamiento de m, no del DAO.

Por lo tanto, en una prueba Mock se debe crear un objeto falso (simulado) que devuelva lo que se necesita para la prueba.

Isolated (Aislado)

Jamás se deben diseñar pruebas que dependan de otros casos de prueba. No importa con qué cuidado se diseñen los casos de prueba, estos pueden contener errores y se puede gastar más tiempo averiguando qué caso de prueba fallo en la cadena de dependencia.

Para garantizar que una prueba es independiente, las pruebas se deben enfocar en cantidades pequeñas de comportamiento.

Repeatable (Repetible)

Una prueba repetible es aquella que produce el mismo resultado cada vez que se ejecuta, para ello se debe aislar la prueba de cualquier entorno externo que no esté bajo el control de la aplicación (utilizar objetos mock).

Si una prueba no se puede repetir, entonces se estará gastando tiempo en perseguir eventos fantasma que pueden o no volverse a presentar.

Self-validating (Validación propia)

Cada prueba debe ser capaz de validar si la salida es la esperada o no, sin necesidad de una interpretación manual de resultados. Se deben establecer todos los requerimientos de las pruebas de tal manera que se puedan ejecutar automáticamente.

Tener datos precargados para alimentar una prueba posibilita ejecutar ene veces la prueba sin que algún factor externo pueda afectar la ejecución y el resultado de la misma.

Timely (En tiempo)

Se pueden crear pruebas en cualquier momento, ya sea cuando la aplicación está a punto de salir a producción o cuando está recién liberada por los desarrolladores.

Es recomendable tener directrices o reglas estrictas para generar las pruebas unitarias, realizando procesos de revisión o utilizando herramientas automáticas para rechazar código que no contenga las pruebas suficientes.



DISEÑO ORIENTADO A PRUEBAS



Dentro de la ingeniería de software todo parte de una idea inicial, un área de oportunidad dentro de las reglas de un negocio que puede ser optimizada a partir de una aplicación de software.

Cuando un cliente ve que una regla de negocio se puede sistematizar se lleva a cabo el levantamiento de requerimientos para generar los requerimientos iniciales que satisfagan las necesidades del negocio. Estos requerimientos dictan el qué se debe realizar.



Posteriormente, estos requerimientos son llevados al proveedor (o equipo desarrollador) para que éste genere una solución. Cuando se diseña la solución se entra en la etapa de cómo satisfacer los requerimientos del usuario, es decir, cómo debe actuar la aplicación para cumplir con las reglas de negocio.

Después viene la etapa de desarrollo donde, a partir del diseño de la solución, se codifica la aplicación en un lenguaje y plataforma específicos.



Ya que se tiene la aplicación desarrollada se procede a probar las reglas del negocio (requerimientos funcionales) y las condiciones establecidas por el cliente (requerimientos no funcionales).

Sin embargo, ¿qué pasa si el negocio cambia y la aplicación no es suficientemente dinámica para soportarlo? Y las áreas de oportunidad encontradas en el desarrollo o los puntos que no se consideraron en los requerimientos iniciales ¿cómo se pueden atender?



En un enfoque de diseño tradicional estos problemas provocan que se regrese a la parte de diseño de la aplicación para poder incluir todas las mejoras, es decir, caer en el ciclo de vida del software.

¿Existe una alternativa más dinámica?

Test Driven Development

El diseño orientado a pruebas o TDD por sus siglas en inglés es una técnica de diseño e implementación de software que se centra en tres pilares fundamentales:

- **Implementar funciones que el cliente necesita y no más.**
- **Minimizar el número de defectos (errores) que llegan al software en producción.**
- **Producir software modular, reutilizable y preparado para el cambio.**



TDD trabaja con ejemplos, es decir, en lugar de partir de requerimientos escritos por el cliente (los cuales pueden ser confusos y extensos), se genera una solución partiendo del resultado a donde se quiere llegar.

Por ejemplo, en un enfoque tradicional para generar una aplicación que permita registrar asistencias de una empresa, se parte de los requerimientos del cliente. En un diseño orientado a pruebas la aplicación se basa en un ejemplo del registro como se lleva a cabo actualmente por el cliente.

Kent Beck, uno de los padres de la metodología XP, menciona algunas razones por las cuales usar TDD:

- La calidad de software aumenta.
- Código altamente reutilizable.
- Se promueve el trabajo en equipo.
- Genera confianza en el equipo de trabajo.
- Multiplica la comunicación entre los miembros del equipo.
- QA adquieren un rol más inteligente e interesante.
- Escribir el ejemplo antes que el código obliga a describir el mínimo de funcionalidad necesaria.
- Las pruebas son una excelente documentación técnica para entender cada uno de los módulos creados.



La esencia de TDD es sencilla, pero ponerla en práctica es cuestión de entrenamiento. El algoritmo para implementar TDD consta de tres pasos:

- **Escribir la especificación (prueba) del requisito (ejemplo).**
- **Implementar el código basado en el requisito (ejemplo).**
- **Refactorizar para eliminar duplicidad y hacer mejoras.**

Refactoring

El refactoring o refactorización es una técnica disciplinada para reestructurar un bloque de código existente, alterando su estructura interna sin cambiar su comportamiento externo.

Por lo tanto, la refactorización consiste en realizar pequeños cambios de comportamiento (en un bloque de código), disminuyendo la probabilidad de que algo salga mal. El refactoring no altera el funcionamiento del sistema.

El refactoring se debe aplicar cuando el código “huela” (se lea) mal:

- **Bloaters:** Código, métodos o clases que han crecido de manera desmesurada y con los cuales es difícil trabajar. Puede involucrar métodos o clases con muchas líneas de código, una larga lista de parámetros de entrada o obsesión por los datos primitivos.
- **Abusador de la orientación a objetos:** Principios de la programación orientada a objetos incompleta o mal implementada.

- **No anticipador al cambio:** Cuando se requiere hacer un cambio en el código, éste se debe hacer en varios lugares también.
- **Dispensables:** El código contiene elementos que no son necesarios y que, de no estar, harían el código más limpio y entendible. Algunos ejemplos son comentarios de más, código duplicado o código muerto.
- **Acopladores:** Existe un alto acoplamiento entre clases o, en lugar de acoplar, se delegan tareas en exceso (cadena de mensajes).



Algunas técnicas para aplicar refactoring son:

- **Arreglar los métodos:** La mayor parte de la refactorización se logra con componer los métodos. Los métodos largos pueden ser la raíz de todo mal.
- **Mover características entre clases:** Mover de manera segura la funcionalidad entre clases, escondiendo los detalles de la implementación de un acceso público.

- **Simplificar expresiones condicionales:** Las estructuras condicionales suelen volverse más complicadas conforme avanza el tiempo, por lo que se debe analizar su pertenencia o su posible descomposición.
- **Simplificar las llamadas a métodos:** Hacer las llamadas a métodos simples y fáciles de entender.
- **Manejar la generalización:** La abstracción tiene sus propias técnicas de refactorización, principalmente basadas en mover funcionalidades a lo largo de la jerarquía de clases.

5 MANEJO DE EXCEPCIONES Y ERRORES





**“Beware of bugs in the above code; I have only proved it correct,
not tried it.”**

Donald Knuth

**(American computer scientist, mathematician, and professor
emeritus at Stanford University.)**



La ley de Pareto, también conocida como la regla 80-20, establece que el 80 % de las consecuencias proviene del 20 % de las causas.

Una máxima en el desarrollo de software dicta que el 80 % del esfuerzo (en tiempo y recursos) produce el 20 % del código. Así mismo, en términos de calidad, el 80 % de las fallas de una aplicación es producida por el 20 % del código.



Por lo tanto, detectar y manejar errores es la parte más importante en una aplicación robusta.

Una aplicación puede tener diversos tipos de errores, los cuales se pueden clasificar en tres grandes grupos: errores sintácticos, errores semánticos y errores en ejecución.

Errores sintácticos.

Son todos aquellos errores que se generan por infringir las normas de escritura de un lenguaje: coma, punto y coma, dos puntos, palabras reservadas mal escritas, etc.

Normalmente son detectados por el compilador o el intérprete (según el lenguaje de programación utilizado) al procesar el código fuente.

Errores semánticos (o lógicos).

Son errores más sutiles, se producen cuando la sintaxis del código es correcta, pero la semántica o significado no es el que se pretendía.

Los compiladores e intérpretes sólo se ocupan de la estructura del código que se escribe y no de su significado, por lo tanto, un error semántico puede hacer que el programa termine de forma anormal, con o sin un mensaje de error.

No todos los errores semánticos se manifiestan de una forma obvia. Un programa puede continuar en ejecución después de haberse producido errores semánticos, pero su estado interno puede ser distinto del esperado.

Por ejemplo, quizá las variables no contengan los datos correctos, o bien es posible que el programa siga un camino distinto del pretendido. Eventualmente, la consecuencia será un resultado incorrecto.

Estos errores se denominan lógicos, ya que, aunque el programa no se bloquea, la lógica que representan contiene un error.

Errores de ejecución.

Son errores que se presentan cuando la aplicación se está ejecutando.

Su origen puede ser diverso, se pueden producir, por ejemplo, por un uso incorrecto del programa por parte del usuario (si ingresa una cadena cuando se espera un número).



Los errores de ejecución también se pueden presentar debido a errores de programación (acceder a localidades no reservadas o hacer divisiones entre cero), o debido a algún recurso externo al programa (al acceder a un archivo o al conectarse a algún servidor o cuando se acaba el espacio en la pila (stack) de la memoria).

Un error en tiempo de ejecución provoca que la aplicación termine abruptamente. Los lenguajes orientados a objetos proveen mecanismos para manejar errores de ejecución.



5. I DEFINICIÓN Y DIFERENCIA ENTRE ERROR Y EXCEPCIÓN.

Excepción

El término excepción hace referencia una condición excepcional que cuando ocurre altera el flujo normal del programa en ejecución.

Estos errores pueden ser generados por la lógica del programa (como un índice de un arreglo fuera de su rango, una división entre cero) o errores generados por los propios objetos que denuncian algún tipo de estado no previsto o condición que no pueden manejar.



Sin embargo, a pesar de ser condiciones anómalas, pueden ser hasta cierto punto previsibles y, por ende, una excepción puede ser controlada para evitar que el programa termine abruptamente, dando una solución general a la condición suscitada.

Error

El término error hace referencia una condición excepcional que cuando ocurre impide el flujo normal del programa en ejecución.

Un error indica un problema serio que no es necesario capturar y manejar debido a que, por la naturaleza propia de este tipo de eventos, un programa no se puede recuperar de un error.



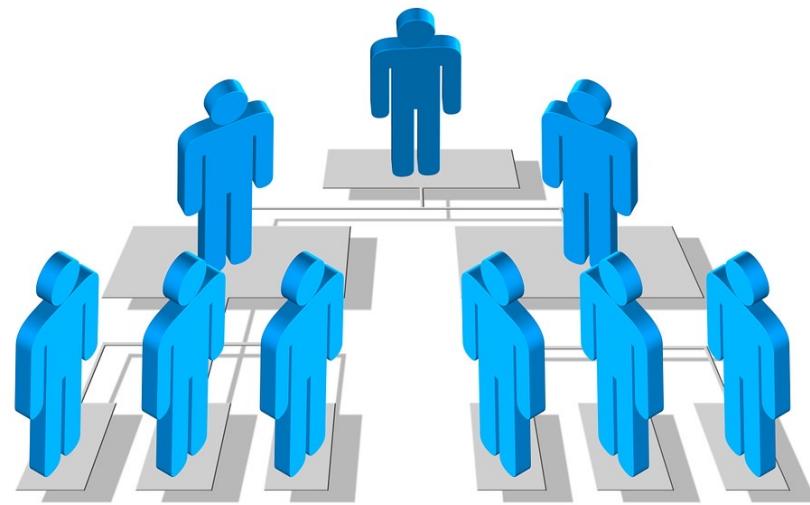
Situaciones como falta de memoria, una clase no encontrada o una versión del compilador no soportada, son errores que impiden el flujo del programa pero que, además, no es posible solucionar en tiempo de ejecución.

Normalmente, cuando ocurre un error la aplicación no se puede recuperar y, por tanto, termina la ejecución del programa.



“No one in the brief history of computing has ever written a piece of perfect software. It’s unlikely that you’ll be the first.”

Andy Hunt
(A writer of books on software development.)



5.2 JERARQUÍA DE CLASES DE ERRORES.

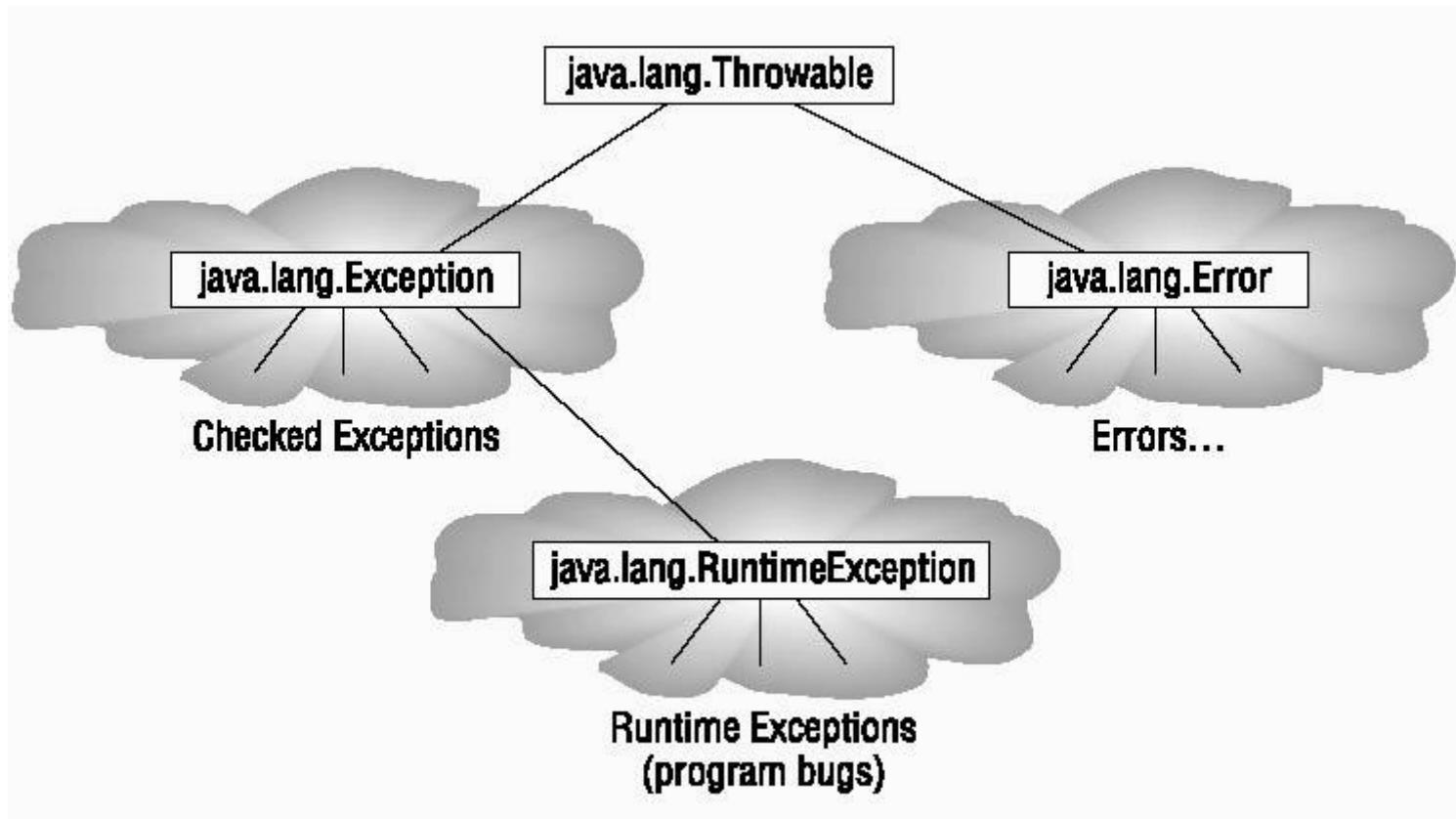
Para tratar una excepción dentro de un programa en Java, hay que tomar en cuenta la jerarquía de clases que permiten manejar excepciones, la cual se dividen en dos grandes grupos:

- **Excepciones marcadas:** Aquellas cuya captura es obligatoria.
- **Excepciones no marcadas:** Las excepciones en tiempo de ejecución (Runtime Exception y sus subclases). No es obligatorio capturarlas.



Las excepciones son objetos que contienen información del error que se ha producido.

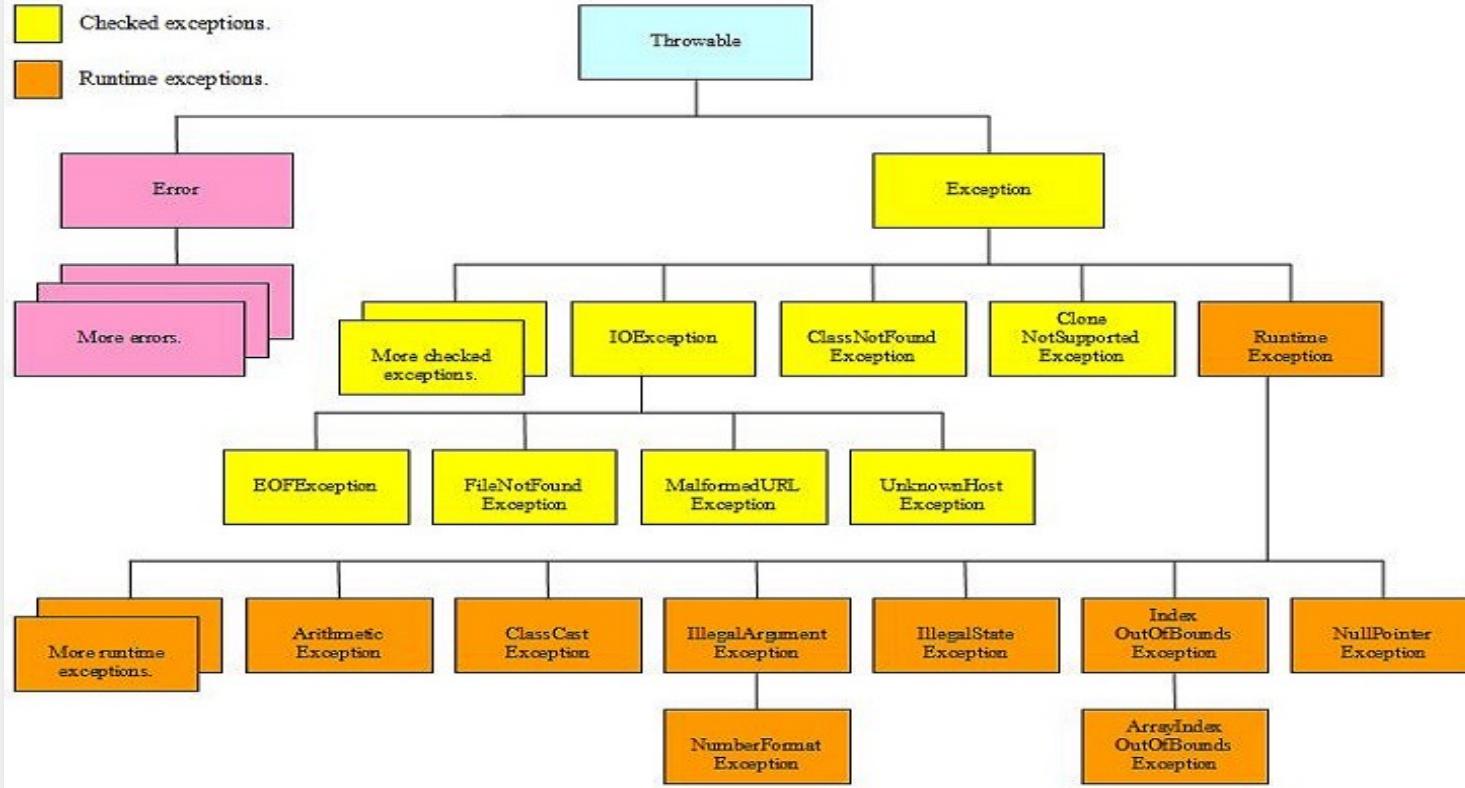
Todas las condiciones excepcionales (Excepciones o Errores) heredan de la clase Throwable que, a su vez, hereda de la clase Object, dibujando así la jerarquía de clases de las excepciones.



Excepciones controladas y no controladas en Java.

Exception Hierarchy

- [Light Blue Box] All exceptions inherit Throwable methods.
- [Pink Box] Errors thrown by the JVM.
- [Yellow Box] Checked exceptions.
- [Orange Box] Runtime exceptions.



Jerarquía de clases de Excepciones y Errores en Java.



5.3 ESTRUCTURA TRY-CATCH-FINALLY.



Para capturar y manejar errores en tiempo de ejecución Java cuenta con la estructura de control try-catch-finally.

Si no se captura la excepción interviene un manejador por defecto que normalmente imprime información que ayuda a encontrar como se produjo el error, sin embargo, como éste no fue capturado, el programa termina abruptamente.

Bloque try

La estructura try-catch-finally está compuesta por tres partes, el bloque try, el bloque catch y el bloque finally.

Dentro del bloque try se coloca el código que se quiere ejecutar y que podría contener un error (división entre cero, índice fuera del arreglo, comparación entre tipos diferentes, acceso a un repositorio de datos, lectura de caracteres, etc.).

Bloque catch

El bloque catch permite capturar cualquier excepción que se presente en tiempo de ejecución a partir del código del bloque try, impidiendo así que el programa deje de ejecutarse y posibilitando al desarrollador enviar el error generado.

Dentro de una estructura try-catch-finally se pueden tener más de un bloque catch, es decir, se pueden capturar las condiciones excepcionales de diferentes tipos (jerarquía de clases).

La única condición para poder tener varios bloques catch dentro de un código es que se definan de las excepciones más específicas hacia las más generales.

```
catch (ClassNotFoundException e) {}  
catch (IOException e) {}  
catch (Exception e) {}
```

Esto es debido a que el error arrojado desde el bloque try va a buscar ser capturado con la excepción que más se le parezca, sin embargo, si se declara primero la excepción más general (Exception), todas los errores siempre caerían ahí.

Bloque finally

El bloque **finally** le indica al programa las instrucciones a ejecutar, de manera independiente de los bloques try-catch.

Si el código del bloque **try** se ejecuta de manera correcta, se ejecuta el bloque **finally**, si dentro del bloque **try** se genera un error, se ejecuta el bloque **catch** y, después, se ejecuta el bloque **finally**. Por lo tanto, se dice que el bloque **finally** siempre se ejecuta.

La sintaxis de la estructura de control try-catch-finally es:

```
try {  
    // bloque de código a ejecutarse  
    // dentro del flujo normal del programa  
} catch (Exception e) {  
    // bloque de código a ejecutarse  
    // si ocurre un error durante el flujo normal  
} finally {  
    // bloque de código a ejecutarse al final  
    // independientemente de si genero  
    // o no una excepción  
}
```

La estructura try-catch-finally puede constar sólo de dos bloques: try-catch o try-finally.

```
try {  
    // bloque de código a ejecutarse  
    // dentro del flujo normal del programa  
} catch (Exception e) {  
    // bloque de código a ejecutarse  
    // si ocurre un error durante el flujo normal  
}
```

Esta estructura permite capturar el error y evita que el programa terminte, sin embargo, no proporciona certeza o modularidad.

La estructura try-catch no proporciona certeza debido a que no se puede asegurar que todo el bloque try se va a ejecutar y tampoco se puede asegurar que siempre va a ocurrir una excepción y, por ende, va a ejecutarse el bloque catch.

```
try {  
    1. Abre una conexión  
    2. Realiza una consulta, se puede generar una excepción  
} catch (Exception e) {  
    3. Error al abrir o realizar la consulta  
    4. Se cierra la conexión  
}
```

No se puede asegurar que 1 y 2 se ejecuten correctamente, sin embargo, si lo hacen, 4 nunca se va a ejecutar.

La estructura try-catch-finally puede constar sólo de dos bloques: try-catch o try-finally.

```
try {  
    // bloque de código a ejecutarse  
    // dentro del flujo normal del programa  
} finally {  
    // bloque de código a ejecutarse al final  
    // independientemente de si genero  
    // o no una excepción  
}
```

Esta estructura provee la certeza de que siempre se va a ejecutar el bloque finally, sin embargo, no captura el error.

La estructura try-finally no captura el error, lo que provocaría que el programa, después de ejecutar el bloque finally, termine abruptamente.

```
try {  
    1. Abre una conexión  
    2. Realiza una consulta, se puede generar una excepción  
} finally {  
    3. Se cierra la conexión  
}
```

No se puede asegurar que 1 y 2 se ejecuten correctamente, pero 3 siempre se va a ejecutar. Sin embargo, si 1 o 2 lanzan una excepción después de ejecutar 3 el programa terminá.

```
public class TryCatchFinally {  
    public static void main(String[] args) {  
        try {  
            String mensajes[] = {"Primero", "Segundo", "Tercero" };  
            for (int i=0; i<=3; i++) {  
                System.out.println(mensajes[i]);  
            }  
        } catch ( ArrayIndexOutOfBoundsException e ) {  
            System.out.println("\nError: índice fuera del arreglo.");  
        } finally {  
            System.out.println("\nEl bloque finally siempre se ejecuta.");  
        }  
        System.out.println("\nContinua el flujo del programa.");  
    }  
}
```



5.4 MANEJO DE ERRORES Y EXCEPCIONES.

Propagación de excepciones

Como se mencionó anteriormente, en el bloque try se ingresa el código que puede ser propenso a enviar una excepción. Sin embargo, no es obligatorio tratar las excepciones dentro de un bloque manejador de excepciones, pero, en tal caso, se debe indicar explícitamente a través del método.

Es decir, un método puede tratar una excepción a través de la estructura try-catch-finally, o puede arrojar la excepción generada. A esta última acción se le conoce como propagación del error.

throws

En Java, un método que puede arrojar (propagar) una excepción debe hacerlo de manera explícita en la firma del método, utilizando la palabra reservada throws, utilizando la siguiente sintaxis

```
[modificadores] valorRetorno nombre () throws Expcion1, Expcion2 {  
    // Bloque de código del método  
}
```



La sintaxis anterior obliga a llamar al método dentro de un manejador de excepciones (try-catch-finally) o dentro de un método que indique que va a arrojar la misma excepción, propagando el error.

throw

Una excepción se puede lanzar de manera explícita (sin necesidad de que ocurra un error) creando una instancia de la misma y arrojándola mediante la palabra reservada `throw`.

Una excepción se debe arrojar dentro de un manejador de excepciones o dentro de un método que indique que se va arrojar dicha excepción (o cualquier subclase). La sintaxis es la siguiente:

```
throw new Expcion();
```

Ejemplo 2

```
public class A{  
    private B be;  
  
    public void setB(B be){  
        this.be = be;  
    }  
  
    public B getB(){  
        return this.be;  
    }  
  
    public void queryA() {  
        try{  
            be.setCe(new C());  
            be.queryBe();  
        } catch (Exception e){  
            System.out.println("Error message: " + e.getMessage());  
            System.out.println("Traza del error:");  
            e.printStackTrace();  
        }  
    }  
}
```

Ejemplo 2

```
public class B{  
    private C ce;  
  
    public void setCe(C ce){  
        this.ce = ce;  
    }  
  
    public C getCe(){  
        return this.ce;  
    }  
  
    public void queryBe() throws Exception{  
        ce.queryCe();  
    }  
}
```

Ejemplo 2

```
public class C{  
    public void queryCe() throws Exception{  
        throw new Exception("Excepción generada en query Ce");  
    }  
}
```

Los métodos `getMessage` y `printStackTrace` se heredan de `Throwable`. El método `getMessage` imprime la cadena que se envía al objeto cuando se construye. El método `printStackTrace` imprime la traza del error, es decir, el método donde se generó el problema y todos los métodos que se invocaron ante de éste.

Ejemplo 2

```
public class TestA {  
    public static void main (String [] args){  
        A a = new A();  
        a.setB(new B());  
        a.queryA();  
    }  
}
```

Excepciones propias

Cuando se desarrollan aplicaciones es común requerir excepciones que no están definidas en el API de Java, es decir, excepciones propias del negocio.

Es posible crear clases que se comporten como excepciones (que se puedan arrojar), para ello sólo es necesario heredar de `Exception` o de `Throwable`.



Las excepciones propias se invocan y se pueden arrojar de la misma manera en la que se utilizan las excepciones del API, pero pueden generar información más concisa del problema debido a que se están programando a la medida del negocio.

Ejemplo 3

Se tiene una aplicación que requiere un inicio de sesión. Cuando el usuario falle tres veces seguidas su nombre de usuario o contraseña, el sistema debe enviar un error y bloquear el acceso del usuario.

Ejemplo 3

```
public class LogInException extends Exception {  
    public LogInException() {  
        super("Falló el inicio de sesión.");  
    }  
  
    public LogInException(String message){  
        super(message);  
    }  
}
```

Ejemplo 3

```
public class LogIn {  
    private byte attempt;  
  
    public boolean logIn(String user, String pwd) throws LogInException{  
        if (user.equals("root") && pwd.equals("gnuLinux")){  
            return true;  
        } else {  
            attempt++;  
            if (attempt == 3){  
                throw new LogInException("Número máximo de intentos.");  
            }  
            return false;  
        }  
    }  
}
```

Ejemplo 3

```
import java.util.Scanner;
public class TestLogIn {
    public static void main (String [] arg){
        LogIn enter = new LogIn();
        while (true){
            try {
                Scanner read = new Scanner(System.in);
                System.out.println("Usuario:");
                String user = read.nextLine();
                System.out.println("Contraseña:");
                String pwd = read.nextLine();
                boolean res = enter.logIn(user, pwd);
                if (res){
                    System.out.println("Bienvenido al sistema " + user + ".");
                } else {
                    System.out.println("Error al iniciar sesión.");
                }
            } catch (LogInException e){
                System.out.println(e.getMessage());
                System.out.println("Su cuenta ha sido bloqueada.Au revoir!");
                break;
            }
        }
    }
}
```

Sobrescritura de excepciones

Para poder sobrescribir un método que arroja una excepción, el método que sobrescribe debe seguir las siguientes reglas:

- Puede no arrojar excepciones.
- Puede arrojar una o más de las excepciones que declara el método base.
- Puede arrojar una o más subclases de las excepciones que declara el método base.



Lo que no puede hacer un método que sobrescribe otro que arroja una excepción es:

- Agregar excepciones que no arroja el método base.
- Agregar súper clases de las excepciones declaradas en el método base.

```
public class LogInSocialNet extends LogIn {  
    private byte attempt;  
  
    // Error al sobrescribir el método LogIn  
    public boolean logIn(String user, String pwd) throws Exception{  
        if (user.equals("root") && pwd.equals("gnuLinux")){  
            return true;  
        } else {  
            attempt++;  
            if (attempt == 3){  
                throw new LogInException("Número máximo de intentos.");  
            }  
            return false;  
        }  
    }  
}
```



“Testing can show the presence of errors, but no their absence.”

E.W. Dijkstra

(Was a Dutch systems scientist, programmer, software engineer, science essayist, and early pioneer in computing science.)

ASSERTIONS

Ambientes de ejecución

Las empresas de desarrollo de software, normalmente, tienen a disposición tres infraestructuras, conocidas como ambientes, los cuales son: desarrollo, calidad y producción.

En el ambiente de desarrollo, como su nombre lo indica, es donde se genera (desarrolla) el software de la aplicación que se solicita (requiere).



El ambiente de calidad (también conocido como pruebas o preproducción) es donde se copian todos los componentes requeridos (archivos class, txt, sql, xml, etc.) desde el ambiente de desarrollo, para poder probar la calidad (cantidad de errores) del software desarrollado.

El ambiente de producción es el ambiente operativo, es el lugar donde el usuario final va a interactuar con el sistema y el cual debe estar libre de errores.

Assertions

Las assertions o afirmaciones permiten documentar y probar suposiciones programadas, impidiendo continuar con la ejecución de un programa si éste carece de sentido.

Las assertions se recomiendan para ambientes de desarrollo y calidad, nunca para ambientes productivos, debido a que validar las afirmaciones conlleva más tiempo de ejecución, además, en teoría, ya fue probado el código de manera exhaustiva y ya no es posible que esas condiciones se presenten en producción.



Por ejemplo, para realizar una serie de tareas programadas se requiere de una estructura que permita ordenar las tareas por prioridades para así atenderlas. Para tal efecto se puede utilizar una Pila.

Sin embargo, qué pasa si la pila viene vacía, ¿tiene sentido continuar con la ejecución del programa? Es en estas situaciones donde se pueden utilizar las assertions.

Para definir una assertion en Java se utiliza la siguiente sintaxis:

```
assert <expresión booleana> ;  
assert <expresión booleana> :<detalles de la expresión> ;
```

De tal manera que si la expresión booleana se evalúa como falsa, se arroja un AssertionError. Los detalles de la expresión se convierte a String y se utiliza como un texto descriptivo del AssertionError.



Las assertions se recomiendan para documentar y verificar las suposiciones y la lógica interna de un método:

- **Invariantes internos.**
- **Invariantes en el control de flujo.**
- **Condiciones posteriores de un objeto.**
- **Condiciones invariantes de una clase.**



Un uso inapropiado de las assertions son:

- **Validar los parámetros de un método público.**
- **Validar una assertion con un método, ya que puede causar efectos secundarios.**

Invariantes internos

Para realizar una división se requiere que el divisor sea diferente de cero, si es igual a cero no se debe computar:

```
if (divisor > 0) {  
    // hacer esto  
} else {  
    assert (divisor < 0)  
    // hacer esto si divisor < 0  
}
```

Invariantes en el control de flujo

Una enumeración que maneje los puntos cardinales no puede tener otro valor que no sean los definidos en ella:

```
switch (cardinalPoint) {  
    case CardinalPoints.NORTH:  
        break;  
    case CardinalPoints.SOUTH:  
        break;  
    case CardinalPoints.EAST:  
        break;  
    case CardinalPoints.WEST:  
        break;  
    default: assert false :“Punto cardinal desconocido.”  
        break;  
}
```

Condiciones posteriores de un objeto o una clase

Si una pila no disminuye el tope si no que, al contrario, lo incrementa, eso representa una condición inesperada:

```
public Object pop(){
    int position = getTope();
    if (position == 0){
        throw new Exception ("No se puede hacer pop.");
    }
    Object res = getElement(position);
    // Condición posterior
    assert (this.getTope() <= MAX-1);
    return res;
}
```

```
public class MyStack {  
    private final byte MAX = 5;  
    private byte tope;  
    private Object [] stack;  
    public MyStack(){  
        stack = new Object[MAX];  
        for (int cont=0 ; cont<MAX ; cont++){  
            stack[cont] = new Object();  
        }  
        tope = 4;  
    }  
    public byte getMax(){  
        return MAX;  
    }  
    public byte getTope(){  
        return tope;  
    }  
}
```

Ejemplo 5

```
public Object getElement(int position){  
    tope++;  
    return stack[position];  
}  
  
public Object pop() throws Exception{  
    int position = getTope();  
    if (position < 0){  
        throw new Exception ("No se puede hacer pop.");  
    }  
    Object res = getElement(position);  
    // Condición posterior  
    assert (this.getTope() <= MAX-1);  
    return res;  
}  
}
```

Ejemplo 5

```
public class TestMyStack {  
    public static void main (String [] args) throws Exception {  
        MyStack referencia = new MyStack();  
  
        for (int cont=0 ; cont<referencia.getMax() ; cont++){  
            Object obj = referencia.pop();  
            System.out.println(obj);  
        }  
    }  
}
```

Por defecto las assertions están deshabilitadas al compilar un programa, es decir, el bytecode no las incluye y, por tanto, el código corre tan rápido como si estas condiciones no estuviesen escritas.

Para habilitar las assertions se debe especificar al ejecutar el programa de la siguiente manera:

```
java -enableassertions MyClass  
java -ea MyClass
```

Las assertions se encuentran disponibles desde la versión 4 de Java, por lo tanto, si se ejecuta un código con una versión anterior assert no se reconoce como palabra reservada:

```
int assert = getTope();
if (assert == MAX-1) {
    throw Exception ("Error en el índice de la Pila.");
}
```

Para compilar un código con una versión anterior de Java se puede especificar a través de la bandera source:

```
javac -source 1.3 MyStack.java
```

ERROR



Debido a que un error es una situación de la cual normalmente la aplicación no se puede recuperar, los errores no son obligatorios capturarlos.

Un método puede arrojar un error (de la misma manera que puede arrojar una excepción), sin embargo, el método que invoca no está obligado ni a capturar la excepción ni a propagar la misma.

Ejemplo 6

```
public class MyError extends Error {  
    public MyError (){}  
    public MyError (String message){  
        super(message);  
    }  
}
```

Ejemplo 6

```
public class ThrowError {  
    public void throwError () throws MyError {  
        throw new MyError("My own error!");  
    }  
}
```

Ejemplo 6

```
public class TestThrowError {  
    public static void main (String [] error){  
        ThrowError te = new ThrowError();  
        te.throwError();  
    }  
}
```

Ejemplo 6

```
public class ExtendsThrowError extends ThrowError {  
    public void throwError () throws Error {  
        throw new MyError("My own error!");  
    }  
}
```



Checked & unchecked

Desde el punto de vista del tratamiento de una excepción dentro de un programa, las excepción y los errores se dividen en dos grandes grupos:

- **Marcadas (checked):**Aquellas cuya captura es obligatoria.
- **No marcadas (unchecked):** Aquellas cuya captura no es obligatorio.

En Java **Exception** y todas sus subclases pertenecen al grupo de las checked (Excepto **RuntimeException** y todas sus subclases) y, por lo tanto, se debe capturar o propagar la excepción.

Así mismo, **Error** y todas sus subclases y **RuntimeException** y todas sus subclases pertenecen al grupo de las unchecked. Las clases unchecked no es necesario ni capturarlas ni propagarlas.

Ejemplo 7

```
public class ThrowRuntimeException {  
    public void throwRuntimeException () throws RuntimeException {  
        throw new RuntimeException("My own error!");  
    }  
}
```

Ejemplo 7

```
public class TestThrowRuntimeException {  
    public static void main (String [] error){  
        ThrowRuntimeException te = new ThrowRuntimeException();  
        te.throwRuntimeException();  
    }  
}
```

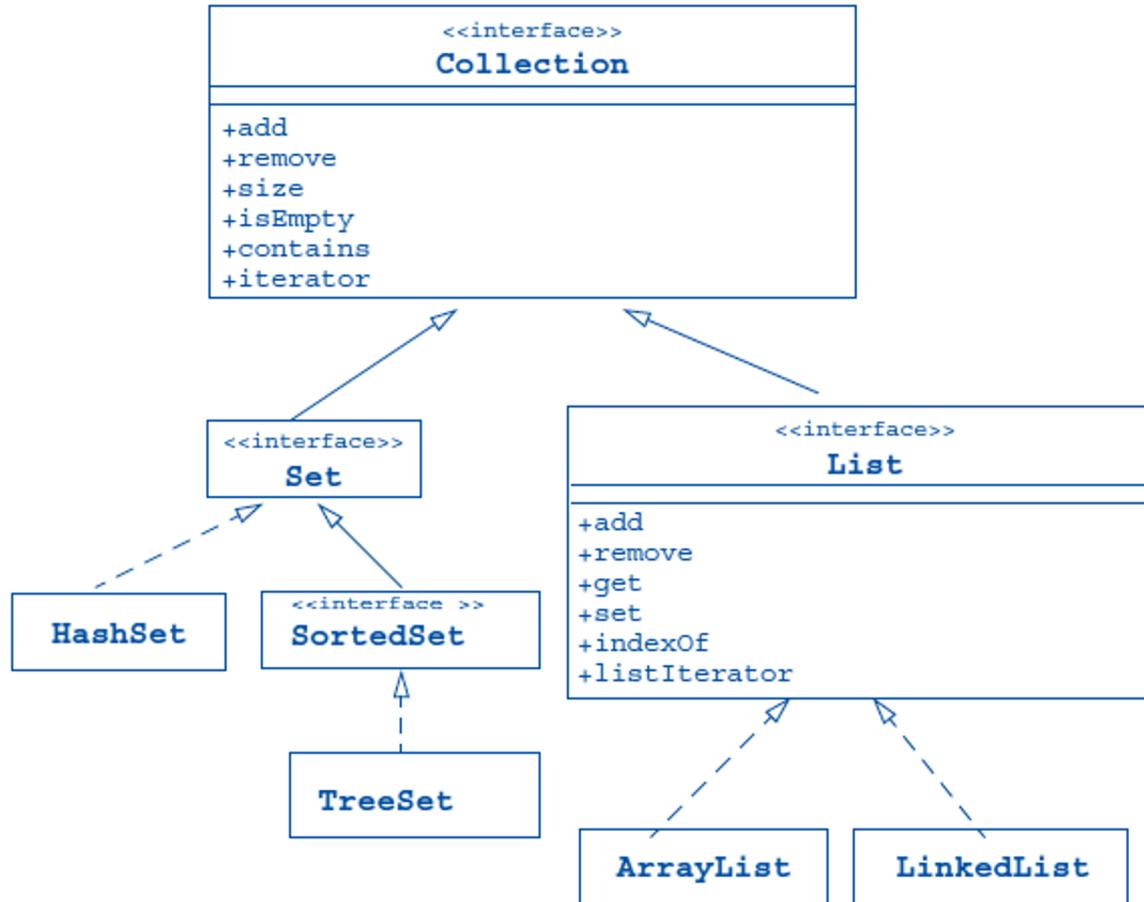


COMPARATOR Y COMPARABLE

COLLECTIONS



Una colección es un conjunto de datos, es decir, una referencia que gestiona un grupo de objetos, a este grupo de objetos se les conoce como elementos de la colección.



Jerarquía de clases de las colecciones.

Interfaz Comparable

La interfaz *Comparable* se utiliza para ordenar colecciones. Provee un orden natural a las clases que la implementan. Es utilizada para ordenar.

Solo posee un método que se debe sobrescribir para proveer un parámetro de comparación:

```
public int compareTo(Object ob)
```

Como se puede observar, el método `compareTo` regresa un valor entero, el cuál puede ser negativo, positivo o cero:

- **Cero:** significa que los elementos comparados son iguales.
- **Positivo (> 0):** significa que el objeto que compara (`this`) es más grande y, por tanto, se debe insertar después del objeto con el que se compara (`Object`).
- **Negativo (< 0):** significa que el objeto que compara (`this`) es menor y, por tanto, se debe insertar antes del objeto con el que se compara (`Object`).

```
public class PersonComparable implements Comparable {  
    private String nombre;  
    private int edad;  
  
    PersonComparable (String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    public int compareTo(Object obj){  
        return this.nombre.compareTo  
            (((PersonComparable)obj).nombre);  
    }  
  
    public String toString() {  
        return nombre + ":" + edad + "";  
    }  
}
```

```
import java.util.TreeSet;

public class TestPerson {
    public static void main (String [] args) {
        TreeSet grupo = new TreeSet ();
        grupo.add(new PersonComparable("Eduardo",18));
        grupo.add(new PersonComparable("Jaime",19));
        grupo.add(new PersonComparable("Alan",17));
        grupo.add(new PersonComparable("Samuel",18));
        grupo.add(new PersonComparable("adriana",18));
        System.out.println(grupo);
    }
}
```

Modificar el método `compareTo` de la clase `PersonaComparable` de tal modo que ordene por edad en lugar de ordenar por nombre.

Interfaz Comparator

La interfaz Comparator representa una relación de orden. Es utilizada para ordenar colecciones. Se utiliza cuando los objetos con los que se trabaja no tienen implementada la interfaz Comparable.

Solo posee un método que se debe sobrescribir para proveer un parámetro de comparación:

```
public int compare(Object ob1, Object ob2)
```

```
public class Student {  
    private String name;  
    private int number;  
    public Student (String name, int number){  
        this.name = name;  
        this.number = number;  
    }  
    public String getName(){  
        return name;  
    }  
    public int getNumber(){  
        return number;  
    }  
    public String toString(){  
        return name + "-" + number;  
    }  
}
```

Ejemplo 9

```
import java.util.Comparator;

public class CompareName implements Comparator {
    public int compare(Object obj1, Object obj2){
        Student s1 = (Student)obj1;
        Student s2 = (Student)obj2;
        return s1.getName().compareTo(s2.getName());
    }
}
```

```
import java.util.Comparator;
import java.util.TreeSet;

public class TestStudent {
    public static void main (String [] args) {
        Comparator c = new CompareName();
        TreeSet grupo = new TreeSet(c);
        grupo.add(new Student("Eduardo",3018));
        grupo.add(new Student("Jaime",3019));
        grupo.add(new Student("Alan",3017));
        grupo.add(new Student("Samuel",3022));
        grupo.add(new Student("adriana",3015));
        System.out.println(grupo);
    }
}
```



PROGRAMACIÓN GENÉRICA

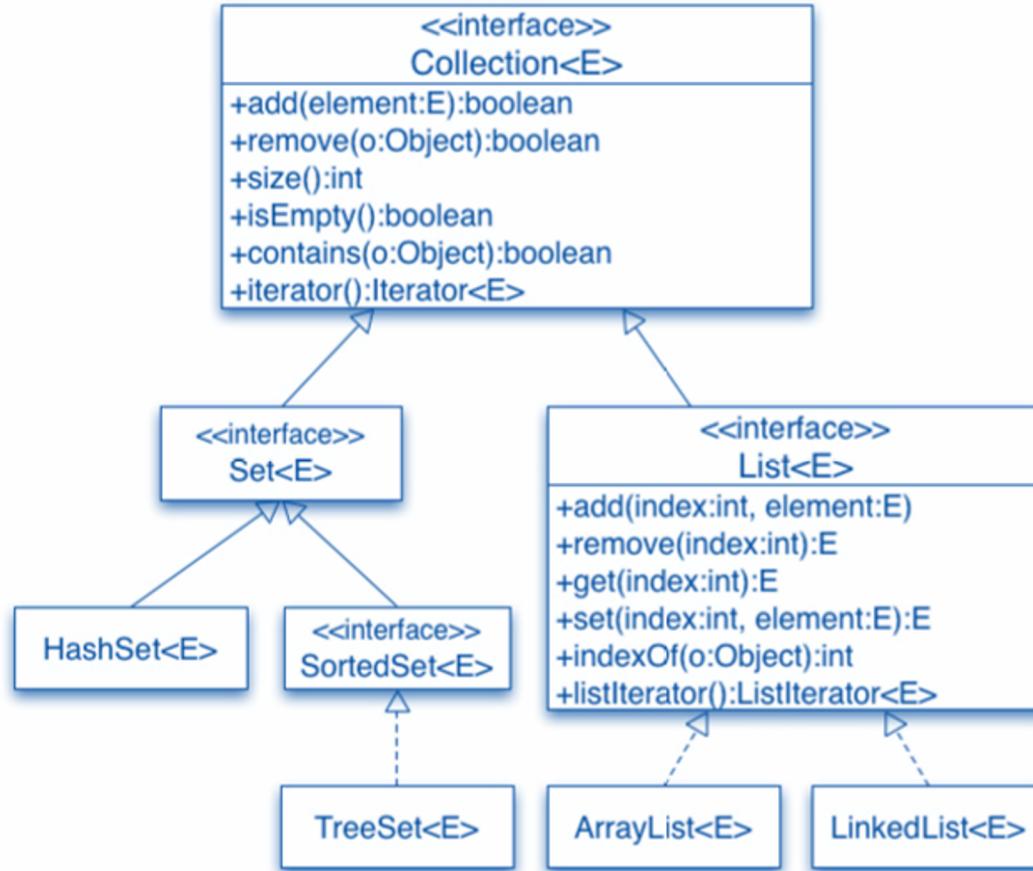
Cuando se crea una colección sin especificar el tipo de dato que va a contener se dice que es una colección no genérica.

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

Cuando se crea una colección especificando el tipo de dato que va a contener se dice que es una colección genérica.

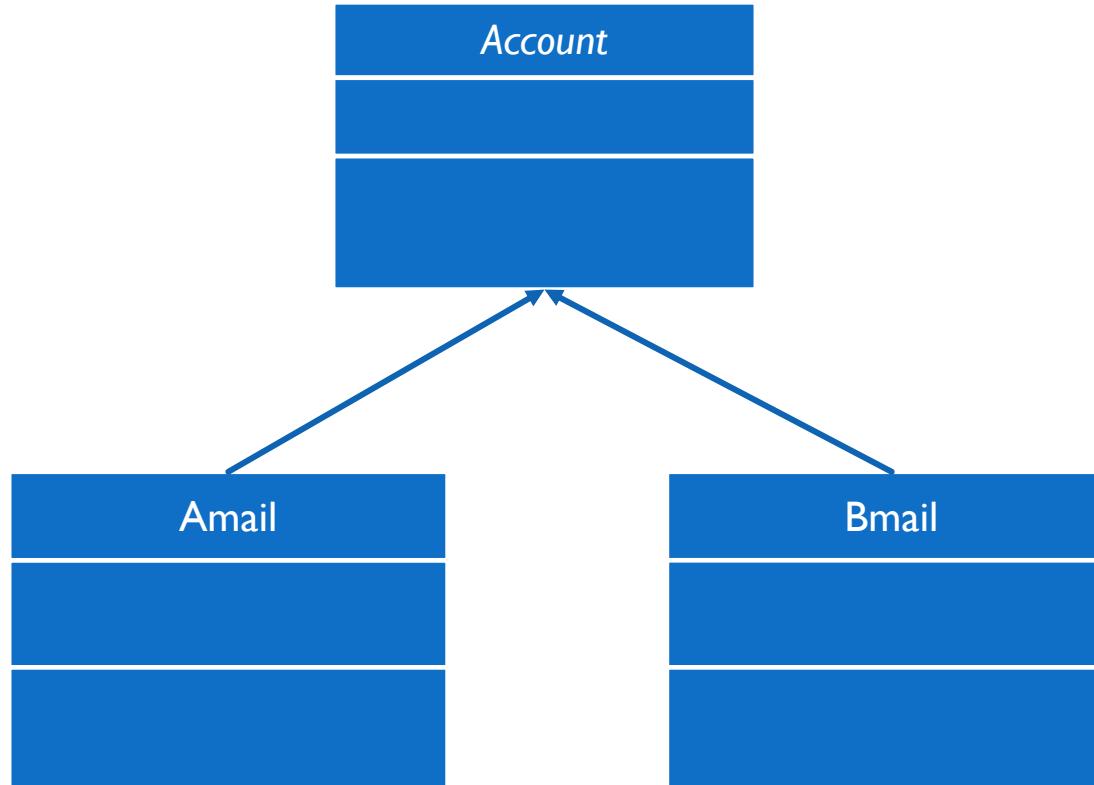
```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(0, new Integer(42));  
int total = (list.get(0)).intValue();
```

```
import java.util.Set;
import java.util.HashSet;
public class GenericSet {
    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();
        set.add("one");
        set.add("second");
        set.add("3rd");
        // This line generates compile error
        //set.add(new Integer(4));
        set.add("second");
        // Duplicate, not added
        System.out.println(set);
    }
}
```



Jerarquía de clases de las colecciones genéricas.

Dada la siguiente jerarquía de clases:



```
import java.util.List;
import java.util.LinkedList;

public class TestGeneric {
    public static void main (String [] arg){
        List<Amail> list = new LinkedList<Amail>();
        list.add(new Amail());
        // Estructura segura en tiempo de compilación
        //list.add(new Bmail());

        // Debido a que es una estructura segura
        // el cast no es necesario
        Amail a = list.get(0);
    }
}
```

```
import java.util.List;
import java.util.LinkedList;

public class TestGeneric {
    public static void main (String [] arg){
        // Una lista de una clase derivada no puede
        // ser asignada a una lista de una súper clase
        List <Account> listAccount = new LinkedList<Account>();
        List <Amail> listA = new LinkedList<Amail>();
        List <Bmail> listB = new LinkedList<Bmail>();
        //listAccount = listA;
        listAccount.add(new Amail());
        //listAccount = listB;
        listAccount.add(new Bmail());
    }
}
```

Ejemplo 10

```
import java.util.List;
import java.util.LinkedList;

public class TestGeneric {
    public static void main (String [] arg){
        List <Account> listAccount = new LinkedList<Account>();
        List <Amail> listA = new LinkedList<Amail>();
        List <Bmail> listB = new LinkedList<Bmail>();
        printList(listAccount);
        printList(listA);
        printList(listB);
    }

    public static void printList(List <? extends Account> list){
        for (int cont=0; cont < list.size() ; cont++){
            System.out.println(list.get(cont));
        }
    }
}
```

Para declarar elementos genéricos se siguen las siguientes reglas:

Categoría	No genérico	Genérico
Clase	public class ArrayList extends AbstractList implements List	public class ArrayList <E> extends AbstractList <E> implements List <E>
Constructor	public ArrayList(int size)	public ArrayList <E> (int size)
Método	public void add (Object o) public Object get (int index)	public void add (E o) public E get (int index)
Variable	ArrayList list;	ArrayList <String> list;
Instancia	list = new ArrayList(5);	list = new ArrayList <String>(5);

```
import java.util.LinkedList;

public class GenericClass <E> {
    LinkedList<E> myList;
    public GenericClass(){
        myList = new LinkedList<E>();
    }

    public void set(E element){
        myList.add(element);
    }

    public E get(int index){
        return myList.get(index);
    }
}
```

```
public static void main (String [] args){  
    GenericClass<String> gc = new GenericClass<String>();  
    gc.set(new String("Hello"));  
    gc.set(new String("Hola"));  
    System.out.println(gc.get(0));  
    System.out.println(gc.get(1));  
}  
}
```

Iterador

La iteración es el proceso de acceder a cada uno de los elementos contenidos en una colección.

En Java, la interfaz Iterator permite recorrer una colección hacia adelante. Además, la interfaz List posee un ListIterator, el cual permite recorrer una colección hacia adelante o hacia atrás.

```
public class Account {  
    public String name;  
  
    public Account(){  
  
    public Account(String name){  
        this.name = name;  
    }  
  
    public String toString(){  
        return this.name;  
    }  
  
    public String getName(){  
        return this.name;  
    }  
}
```

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Collection;

public class TestTypeSafety {

    public static void iterador(Collection <Account> l) {
        System.out.println("Iterator");
        Iterator i = l.iterator();
        while (i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

```
public static void iterador2(Collection <Account> l) {  
    System.out.println("Iterator 2");  
    for (Account a : l) {  
        System.out.println(a);  
    }  
}  
  
public static void iterador3(List <Account> l) {  
    System.out.println("Iterator 3");  
    ListIterator <Account> i = l.listIterator();  
    while (i.hasNext()) {  
        System.out.println(i.next());  
    }  
    while (i.hasPrevious()) {  
        System.out.println(i.previous());  
    }  
}
```

Ejemplo 12

```
public static void main(String[] args) {  
    List<Account> la = new ArrayList<Account>();  
    la.add(new Account("Adrew"));  
    la.add(new Account("Luigi"));  
    la.add(new Account("Fred"));  
    la.add(new Account("Kevin"));  
    la.add(new Account("Laura"));  
    System.out.println(la);  
  
    System.out.println();  
    iterador(la);  
    System.out.println();  
    iterador2(la);  
    System.out.println();  
    iterador3(la);  
}  
}
```

EXPRESIONES LAMBDA

Uno de los conceptos de la programación funcional menciona que las funciones (métodos) pueden ser definidas como entidades de primer nivel, es decir, que pueden aparecer en partes del código donde otras entidades de primer nivel (como valores primitivos u objetos) lo hacen.

Esto significa poder pasar funciones, en tiempo de ejecución, como valores de variables, valores de retorno o parámetros de otras funciones, es decir, pasar el comportamiento de una función como valor. Esto se puede lograr con las expresiones lambda.

Las expresiones (o funciones) lambda permiten referenciar métodos anónimos (sin nombre), se componen de:

- Listado del tipo de parámetros es opcional, el compilador puede inferir el tipo de dato del valor del parámetro.
- Los paréntesis alrededor de un parámetro son opcionales, para más de un parámetro son requeridos.
- El símbolo de fecha a la derecha (->)
- El cuerpo de la función entre llaves para más de una expresión, para una sola expresión no son necesarios.
- El valor de retorno es opcional, el compilador regresa el resultado de la expresión.

Ejemplo 13

```
public interface Message {  
    void getMessage(String message);  
}  
  
public class Regrads {  
    public static void main (String [] args) {  
        Message m = message ->  
            System.out.println("You say: " + message);  
        m.getMessage("Lambda message!!");  
    }  
}
```

```
public interface Operation {  
    int getOperation(int a, int b);  
}  
  
public class Arithmetic {  
    public static void main (String [] args) {  
        Operation o = (a, b) -> {  
            System.out.println("Dentro de lambda");  
            return a + b;};  
        System.out.println(o.getOperation(5, 6));  
    }  
}
```

```
import java.util.ArrayList;
import java.util.List;

public class LambdaExpresions {
    public static void main (String [] args) {
        List<String> lista = new ArrayList<String>();
        lista.add(new String("Hola"));
        lista.add(new String("Adios"));
        lista.add(new String("Bienvenido"));
        lista.add(new String("Au revoir!"));
        System.out.println(lista);
        lista.stream()
            .filter(l -> l.startsWith("A"))
            .map(String::toUpperCase)
            .sorted()
            .forEach(System.out::println);
    }
}
```

5 Manejo de excepciones y errores

Objetivo: Clasificar los diferentes tipos de errores y excepciones para generar programas y aplicaciones con calidad.

5.1 Definición y diferencia entre error y excepción.

5.2 Jerarquía de clases de errores.

5.3 Estructura try-catch-finally

5.4 Manejo de errores y excepciones.