Excepciones

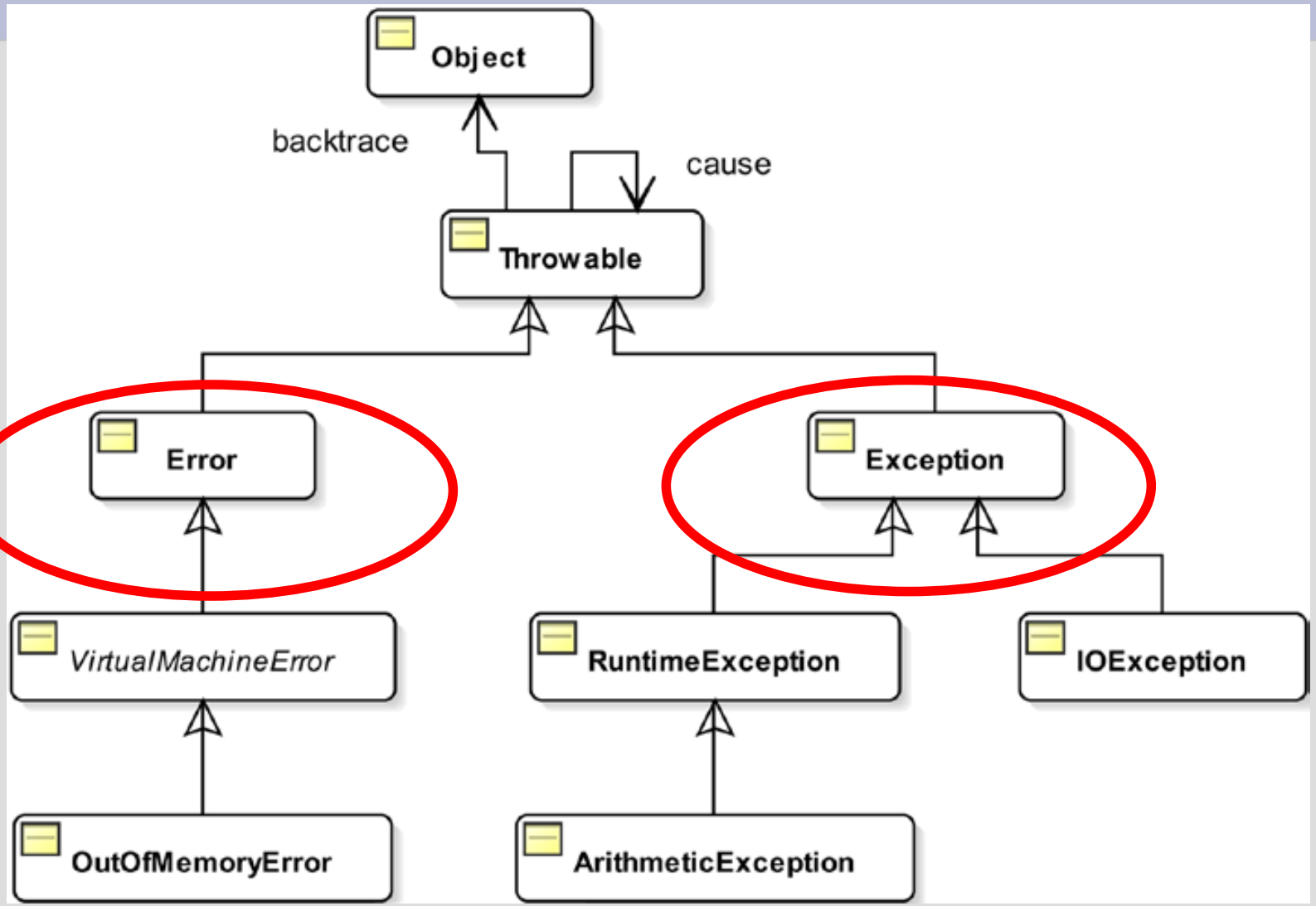
Paradigmas de Lenguajes
3er Año LCC- Facultad CEFyN - UNSJ -

Concepto:

"Una excepción es un evento,
que ocurre durante la ejecución de un programa
y que interrumpe el flujo normal de ejecución"

https://docs.oracle.com/javase/tutorial/essential/exceptions/

Excepciones en Java:



Estas son condiciones excepcionales que son externas a la aplicación y que, por lo general, la aplicación no puede anticipar ni recuperarse.

Por ej: supongamos que una aplicación abre con éxito un archivo para la entrada, pero no puede leer el archivo debido a un mal funcionamiento del hardware o del sistema.

Solo debería ser capturada, para notificar al usuario del problema, pero también **podría tener sentido que el programa imprima un seguimiento de pila y se detenga**.

Excepciones no chequeadas

Son condiciones excepcionales internas de la aplicación y que, por lo general, la aplicación no puede anticipar ni recuperarse.

Son problemas de programación, como errores de lógica o uso incorrecto de una API.

Por ejemplo:

> Considere la aplicación descrita anteriormente que pasa un nombre de archivo al constructor para FileReader. Si un error de lógica hace que se pase un nulo al constructor, el constructor lanzará **NullPointerException**.

La aplicación puede detectar esta excepción, pero probablemente tenga más sentido eliminar el error que provocó la excepción.

Excepciones chequeadas

Estas son condiciones excepcionales que una aplicación bien escrita puede anticipar y luego recuperarse.

Por ej:

Supongamos que una aplicación solicita al usuario un nombre de archivo de entrada y luego abre el archivo pasando el nombre al constructor para **java.io.FileReader**.

Normalmente, el usuario proporciona el nombre de un archivo legible existente, por lo que la construcción del objeto FileReader tiene éxito y la ejecución de la aplicación se realiza normalmente.

Pero a veces el usuario proporciona el nombre de un archivo inexistente y el constructor arroja **java.io.FileNotFoundException.** Un programa bien escrito detectará esta excepción y notificará al usuario, posiblemente solicite ingresar nuevamente el nombre del archivo.

Pauta:

Si es razonable esperar que un cliente se recupere de una excepción, es una excepción chequeada.

Si un cliente no puede hacer nada para recuperarse de la excepción, es una excepción no chequeada.

Manejo de excepciones

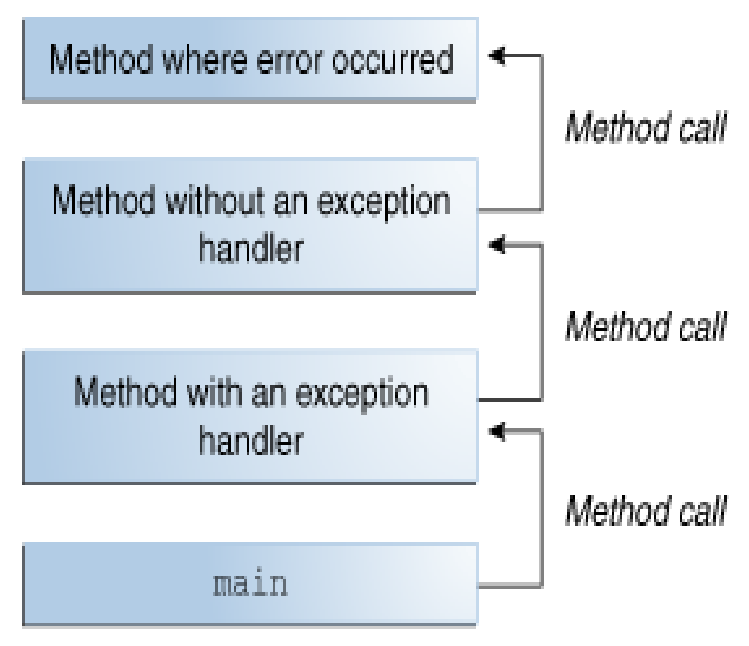
- Cuando se produce un problema en algún método, se crea un objeto, con información, que se transfiere al sistema.

  *Esto es, lanzar – **throw** – la excepción*

- Después de que un método arroja una excepción, el sistema intenta *manejarla* – ***catch***. Para esto sigue la pila de llamadas:

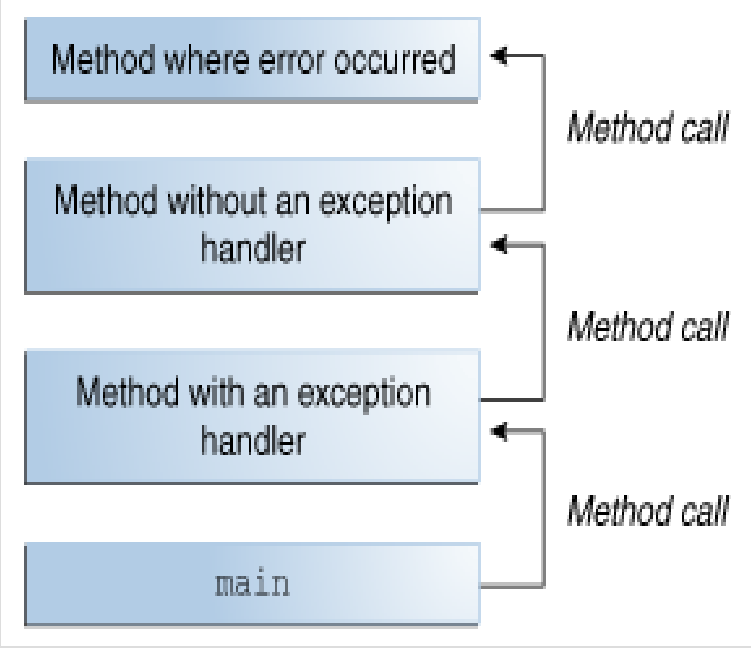
Pila de llamadas:

Pila de llamadas.

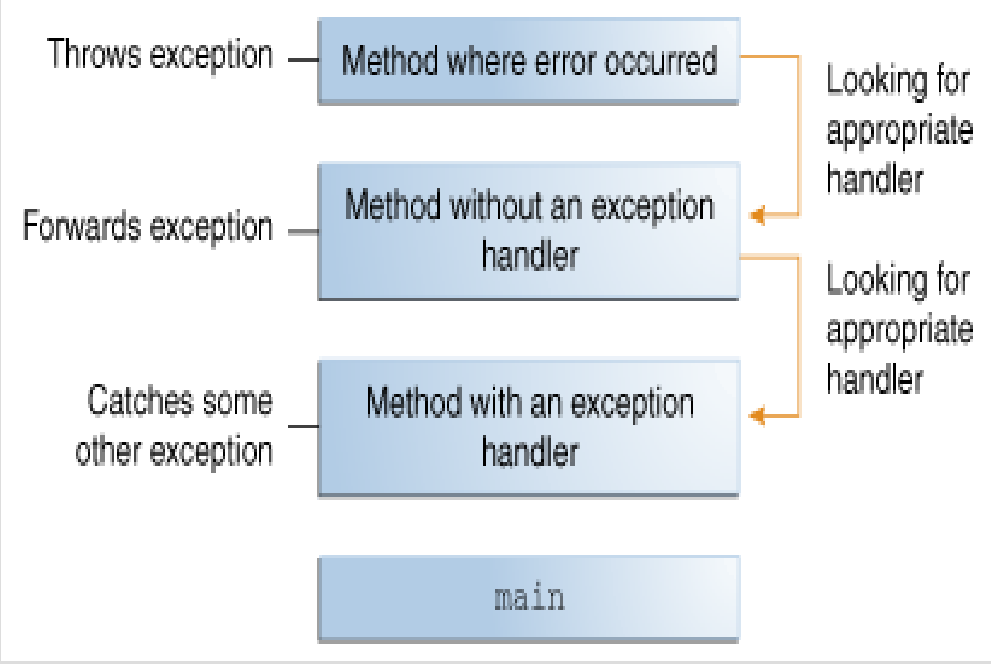


Pila de llamadas:

Pila de llamadas.



Buscando el manejador



Probar:

```
public class Datos {
   static public void main(String[] arg) {
         metodo1();
    }
    public static void metodo1() {
        metodo2();
    }
     public static void metodo2() {
        metodo3();
    }
     public static void metodo3() {
Throwable t = new Throwable();
        StackTraceElement[] frames = t.getStackTrace();
        mostrarPila(frames);
     }
     public static void mostrarPila(StackTraceElement[] frames){
       System.out.println("Canitidad = " + frames.length);
       System.out.println("Nro \tArchivo \tClase \t\t\tMetodo \tLinea" );
       for (int i = 0; i < frames.length; i++) {
           System.out.print(i);
           System.out.print("\t" + frames[i].getFileName());
           System.out.print("\t" + frames[i].getClassName());
           System.out.print("\t" + frames[i].getMethodName());
           System.out.println("\t" + frames[i].getLineNumber());
           System.out.println("-----------------------------------
------");
       }
     }
}
```

Bloque try - catch

1) Para construir un manejador de excepciones, primero se debe incluir el código que podría arrojar (throw) una excepción dentro de un **bloque try**
2) Para asociar un manejador de excepciones con un bloque try, se debe asociar uno o mas **bloques catch**
3) Cada bloque catch es un manejador de excepciones que maneja el tipo de excepción indicado por su argumento, que se corresponde con el nombre de una clase que hereda de la clase **Throwable.**

"No todas las excepciones están sujetas a ser capturadas"

```
package ejemplo;
import java.util.InputMismatchException;
import java.util.Scanner;
public class Datos {
static public void main(String[] arg){
        String nombre;
        int edad;
        Scanner scan = new Scanner(System.in);
        String seguir = "S";
        while(seguir.compareTo("S") == 0) {        // while(seguir.equals("S"))
           try {

       System.out.print("Ingrese nombre: “);
              nombre = scan.next();
              System.out.print("Ingrese edad: “);
              edad = scan.nextInt();
              System.out.println("   Bienvenido "+nombre);
              System.out.print("Ingresa otro dato? (s/n)");
              seguir = scan.next().toUpperCase();

           } catch (InputMismatchException e) {
               System.err.println( scan.next() + " ES INCORRECTO. Intente nuevamente”);
           } finally {    }
        } }}
```

Bloque finally

- El bloque finally siempre se ejecuta cuando sale el bloque try. Esto asegura que el bloque finally se ejecuta incluso si ocurre una excepción inesperada. Pero finalmente es útil para algo más que el manejo de excepciones: permite que el programador evite que el código de limpieza sea anulado accidentalmente por un retorno, continuación o interrupción.
- Poner el código de limpieza en un bloque finally es siempre una buena práctica, incluso cuando no se prevén excepciones.

Nota: Si la JVM sale mientras se está ejecutando el código try o catch, entonces el bloque finally no se puede ejecutar. Del mismo modo, si el subproceso que ejecuta el código try o catch se interrumpe o se cancela, el bloque finally no se puede ejecutar aunque la aplicación como un todo continúe.

Ventajas:

1. Separa el código regular del código de error

La alternativa es introducir muchas declaraciones **if-else anidadas** que dan como resultado código de espagueti. Esto no es elegante ni fácil de mantener. Tener en cuenta que el **código aumenta** considerablemente y las sentencias se mezclan con el código de error.

1. Propagación de los informes de errores en la pila de llamadas.
2. Agrupación y diferenciación de tipos de error

En gral es deseable que los manejadores de excepciones sean lo más específicos posible. La razón es que el controlador debe determinar primero qué tipo de excepción se produjo y luego decidir la mejor estrategia de recuperación. En efecto, al no detectar errores específicos, el controlador debe acomodar cualquier posibilidad. Los manejadores de excepciones que son demasiado generales pueden hacer que el código sea más propenso a errores capturando y manejando excepciones que no fueron anticipadas por el programador y para las cuales el manejador no fue diseñado.

Ejercicios:

1) El siguiente código es legal?

```
try {
 } finally {
}
```

2) Que tipo de excepción puede capturar el manejador?

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

¿Qué hay de malo en usar este tipo de manejador excepciones?

3) ¿Por qué el siguiente código no compila??

```
try {
 // Esta sentencia puede lanzar la excepcion NumberFormatException
 int nro= Integer.parseInt("Hello");
   } catch (Exception e) {
  // Algun codigo
   } catch (NumberFormatException e) {
 // Algun codigo
  }
```

Ejercicios:

4) ¿Se puede lanzar una excepción desde bloque catch? Ej.: throw e;

5) ¿Cuál de las siguientes es una excepción chequeada?

__ java.lang.ArithmeticException

__ java.io.IOException

6) ¿El siguiente código compila? Porque?

```
public void test() {
        throw new RuntimeException("Un error ha ocurrido.");
        System.out.println("Todo superado!!");
}
```

Creando mis excepciones

```
public class Datos {
static public void main(String[] arg){
        System.out.println(“Inscripciones abiertas!!”);
        System.out.println(“Capacitación para jovenes entre 18 y 24 años”);
        String nombre;
        int edad;
        Scanner scan = new Scanner(System.in);
        String seguir = "S";
        while(seguir.equals("S")) {
       System.out.print("Ingrese nombre: “);
              nombre = scan.next();
              System.out.print("Ingrese edad: “);
              edad = scan.nextInt();
              joven(edad);
              System.out.println("   Bienvenido "+nombre);
              System.out.print("Ingresa otro dato? (s/n)");
              seguir = scan.next().toUpperCase();
           }
} }
```

Método que lanza la excepción:

```
public static void joven(int edad) throws MiExcepcion {
        if (edad < 18 || edad > 24) {
            throw new MiExcepcion("No cumple con la edad
                                                requerida");
        }
    }
```

Clase excepción:

```
public class MiExcepcion extends Exception {
     public MiExcepcion() {
super();
     }
    public MiExcepcion(String mensage) {
super(mensage);
    }
    public MiExcepcion(String mensage, Throwable causa) { //
Link
super(mensage, causa);
    }
    public MiExcepcion(Throwable causa) {
super(causa);
    }
}
```

Manejar la excepción:

```
String nombre;
        int edad;
        Scanner scan = new Scanner(System.in);
        String seguir = "S";
        while(seguir.compareTo("S") == 0) {
           try {
              System.out.print("Ingrese nombre: “);
              nombre = scan.next();
              System.out.print("Ingrese edad: “);
              edad = scan.nextInt();
              joven(edad);
              System.out.println("   Bienvenido "+nombre);
              System.out.print("Ingresa otro dato? (s/n)");
              seguir = scan.next().toUpperCase();
           } catch (InputMismatchException e) {
              System.err.println( scan.next() + " ES INCORRECTO. Intente nuevamente”);
            } catch (MiExcepcion e) {
                System.err.println(nombre + " " + e.getMessage());
            }
```

Aserciones

”Permiten detectar errores de lógica durante la etapa de **desarrollo y prueba** del software”

Las aserciones ayudan a los programadores a encontrar rápidamente la ubicación y el tipo de problemas en el código. Una vez que se prueba una aplicación, es muy poco probable que las aserciones fallen.

Ejemplo:

1)

```
int cantidad= 15;
double precioUnitario = getPrecio();
// Aquí estamos seguros que precioUnitario es mayor 0.0
double total = cantidad * precioUnitario;
```

2)

```
// Aquí aseguramos que x es positivo
double y = Math.sqrt(x);
```

Hay dos formas de declaración de aserción:

assert *condicion*;

assert *condicion* : *expresion*;

La instrucción assert evalúa la condición y arroja un **AssertionError** si es falsa.

En la segunda forma, la expresión se convierte en un string que es el mensaje de error.

1)

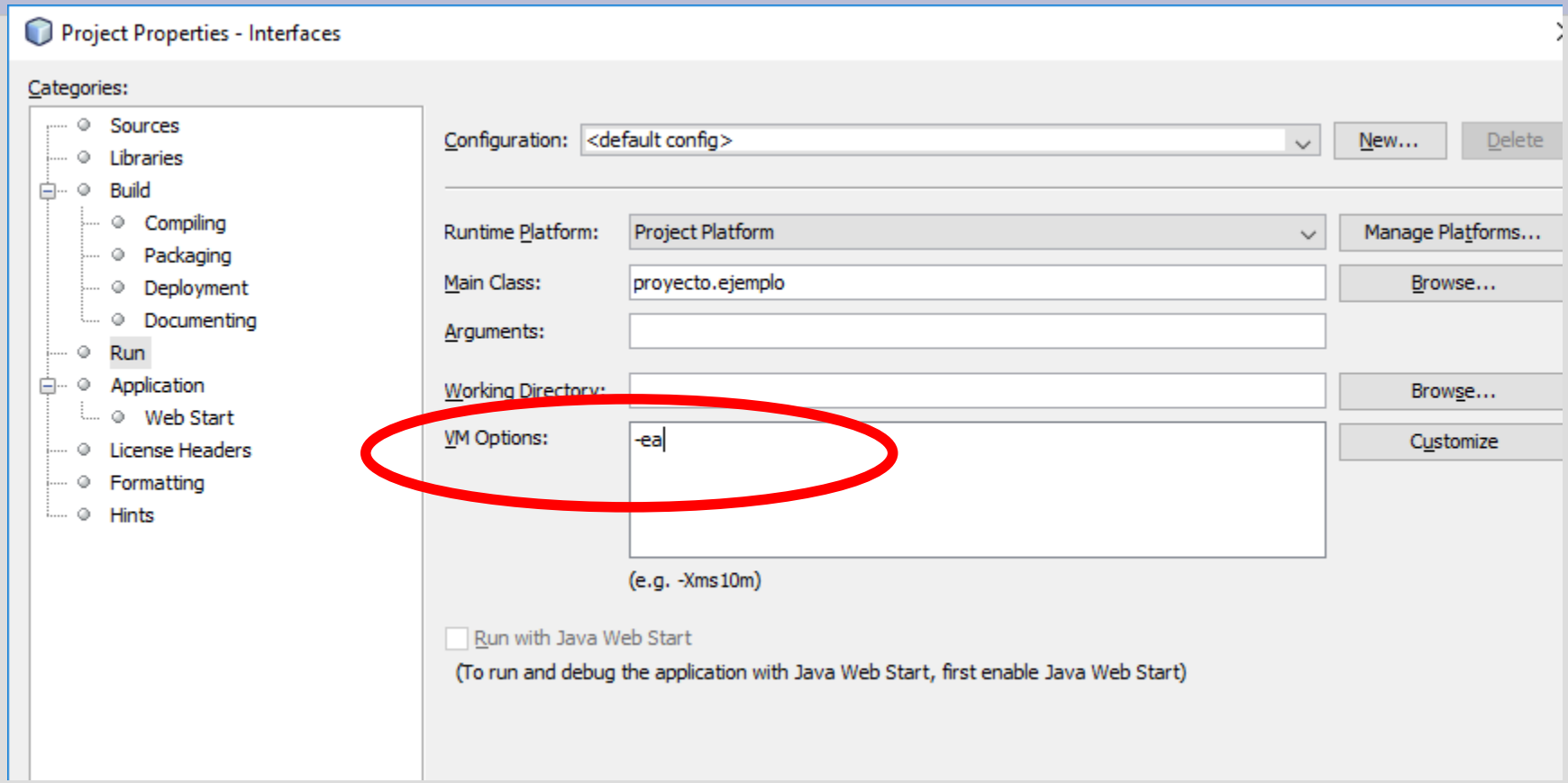
```
int cantidad= 15;
double precioUnitario = getPrecio();
assert  precioUnitario  > 0 : precioUnitario;
// Aseguramos que precioUnitario es mayor 0.0
double total = cantidad * precioUnitrio;
```

2)

```
assert  x > 0 : “x = ” + x;
// Aseguramos que x es positivo
double y = Math.sqrt(x);
```

Por defecto las aserciones están deshabilitadas, esto es por el impacto que tiene en el rendimiento de una aplicación.

Habilitar aserciones:



Desde la línea de comandos:

c:\practica>**java** -jar **-ea** dist/excepcion.jar

Comando	Descripción
-ea	-enableassertions Habilita las aserciones
-ea:MiClase	Se puede habilitar solo para una clase.
-ea:com.mispaquetes...	Se habilita pata todas las clase de un paquete. ... paquete de trabajo actual.
-da	-disableassertions Deshabilita las aserciones
-esa	-enablesystemassertions Habilita aserciones en todas las clases del sistema. No tiene argumento.
-dsa	-disablesystemassertions Deshabilita aserciones en todas las clases del sistema. No tiene argumento.

En Java, las aserciones son una **herramienta de depuración** para validar suposiciones internas, **no** como un mecanismo de validación.

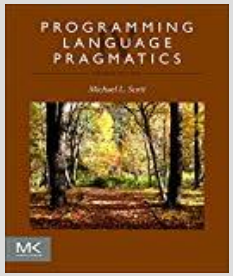
Los diseñadores conocen la penalización de rendimiento cuando se usan aserciones en el entorno de producción.

Por ejemplo, si desea informar un parámetro inapropiado de un método público, no use una aserción sino arroje una IllegalArgumentException

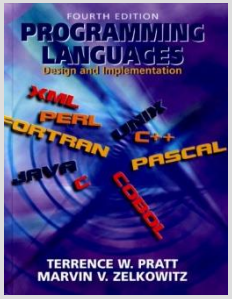
Referencias:

- Sitio Oficial: https://docs.oracle.com/javase/tutorial/essential/exceptions/
- Tutorial de Java: https://www.geeksforgeeks.org/java-tutorials/

Bibliografía:



Programming language pragmatics (4th Edition). 2016. Michael L. Scott.



Programming Languages: Design and Implementation (4th Edition). 2001.Terrence W.PRATT y Marvin V. ZELKOWITZ

Lenguajes de Programación. Diseño e Implementación (3ra. Edición). Terrence W.PRATT y Marvin V. ZELKOWITZ