

## SUBTOPIC 4

# ERROR HANDLING

*Pedro J. Ponce de León*

*Translated into English by Juan Antonio Pérez*

Version 20111005



# Error Handling Objectives

- **Learning how to use exceptions correctly.**
- **Understanding the advantages of exception handling over traditional error handling mechanisms in procedural programming.**
- **Understanding standard exception hierarchy in Java.**
- **Knowing how to create your own exceptions.**
- **Knowing how to handle uncaught and unexpected exceptions.**
- **Differentiating between checked and unchecked exceptions.**

# Contents

- Motivation
- Exceptions
  - Concept
  - Throwing and catching exceptions
  - Defining exceptions
- Standard exceptions in Java
- User exceptions
- Exception features
- Rethrowing an exception

# Error Handling

## Motivation

- Traditional error handling (à la C)

```
int main (void)
{
    int res;
    if (puedo_fallar () == -1)
    {
        cout << "¡Algo falló!" << endl;
        return 1;
    }
    else cout << "Todo va bien..." << endl;
    if (dividir (10, 0, res) == -1)
    {
        cout << "¡División por cero!" << endl;
        return 2;
    }
    else cout << "Resultado: " << res << endl;
    return 0;
}
```

Normal flow  
Error flow

# Error Handling

## Motivation

- This promotes programs with the following structure:
  - Perform task 1
    - *If an error occurred*
      - *Perform error handling*
  - Perform task 2
    - *If an error occurred*
      - *Perform error handling*
- This is called spaghetti code
- Problems with this strategy:
  - Code which describes what you want to do is intermingled with code that is executed when things go wrong (legibility is reduced)
  - The caller is not required to handle the error
  - Error codes are inconsistent across libraries or applications
  - More: **semipredicate problem**

# Error Handling

## Motivation

- Alternatives to spaghetti code:
  - Abort program execution
    - What happens, for instance, with **life-critical** software?
  - Use global error indicators
    - The caller is not required to check these global indicators.
  - USE EXCEPTIONS!
    - Main idea:
      - When enough information to solve a problem in the current context (scope or method) is not available, the caller will get an exception.

# Error Handling

## Exceptions: concept

- An **exception** is a special run-time event that changes the normal flow of program execution. It is important to distinguish an exceptional condition from a normal problem, in which you have enough information in the current context to cope with the difficulty.
- Exceptions are objects which contain information about the error.
- Exceptions are processed by means of sentences which contain the special code for handling the error situation, such as...
  - **throw, try, catch, finally**
- By default, an exception cannot be ignored; as a last resort, it will abort the execution of the program.
  - After creating the exception object, the exception handling mechanism takes over and begins to look for an appropriate place (moving from a method to its caller, and so on) to continue executing the program; this appropriate place is the exception handler, whose job is to recover from the problem so the program can either try another tack or just continue.

# Error Handling

## Exceptions: **behavior**

- A method **throws** an exception when an error condition is detected.
- After throwing an exception, the current path of execution is stopped; if the method does not **catch** the exception, the reference to the exception object is somehow “returned” from the method.
- Any similarity to an ordinary return from a method ends here, because where you return is someplace completely different from where you return for a normal method call. You end up in an appropriate exception handler that might be far—many levels away on the call stack—from where the exception was thrown (the caller, the caller of the caller...).
- If a method throws an exception, it must assume that exception will be **caught** and dealt with, usually by the client code (caller).
- An uncaught exception will terminate the program.



# Error Handling

## Exceptions: **correct use**

```
void f() {  
    try { g(); } catch(Exception ex) {  
        System.err.println(ex.queHaPasado());  
    }  
    // sigue...  
}
```

f() captura el tipo de  
excepciones que puede  
lanzar h()

```
void g() {  
    h();  
    // sigue...  
}
```

g() no la captura...

```
void h() {  
    if (algo_fallar)  
        throw new Exception("¡Mecachis!");  
    // sigue...  
}
```

h() puede lanzar  
una excepción...

### Correct use of exceptions

- The method which throws the exception will not generally catch it. This will be delegated to the callers.

# Error Handling

## Exceptions: **Java syntax**

```
void Func() {  
    if(detecto_error1) throw new Tipo1();  
    ...  
    if(detecto_error2) throw new Tipo2();  
    ...  
}
```



constructor

```
try  
{  
    // Normal flow code  
    Func(); // it may throw exceptions  
}  
catch (Tipo1 ex)  
{  
    // Handle Tipo1 exceptions  
}  
catch (Tipo2 ex)  
{  
    // Handle Tipo2 exceptions  
}  
finally {  
    // Always executed.  
}  
// rest of the code
```

# Error Handling

## Exceptions: C++ syntax

```
void Func() {  
    if (detecto_error1) throw Tipo1();  
    ...  
    if (detecto_error2) throw Tipo2();  
    ...  
}
```



constructor

```
try  
{  
    // Normal flow code  
    Func(); // it may throw exceptions  
    ...  
}  
catch (Tipo1 &ex)  
{  
    // Handle Tipo1 exceptions  
}  
catch (Tipo2 &ex)  
{  
    // Handle Tipo2 exceptions  
}  
catch (...)  
{  
    /* Handle any other exception not  
       caught previously */  
}  
// rest of the code
```

# Error Handling

## Exceptions: **syntax**

- In both C++ and Java:
  - **Try** block contains code which is part of the normal execution flow of the program.
  - **Catch** blocks contain code for error handling.
- Only in Java:
  - **Finally** block is used to close files, free resources, etc. This usually pertains to some operation (cleanup) other than memory recovery (since that is taken care of by the garbage collector). The finally block will execute:
    - (1) after executing the try block or
    - (2) after executing one of the catch clauses
  - Even in cases in which the exception is not caught in the current set of catch clauses, finally will be executed before the exception handling mechanism continues its search for a handler at the next higher level.

# Error Handling

## Exceptions: **behavior**

- Behavior:
  1. Execute the instructions in the try block
    - If an error occurs, stop the execution in the try block and move to the corresponding catch block.
  2. Continue the execution after the catch blocks.
- Exceptions are caught by the catch block whose argument matches the type of the object thrown. Catch blocks are checked in the order defined in the code until the first match is found.
- In case an appropriate handler does not exist, the default mechanism would abort the execution of the program (the exception will be caught and shown by the JVM).

# Error Handling

## Exceptions: **throwing** exceptions

- **throw** instruction is used to raise an exception
  - An exception is an object.
  - Class **Exception** (Java) represents a general exception.

### Throwing

```
int LlamameConCuidado(int x) {  
    if (condicion_de_error(x) == true)  
        throw new Exception("valor "+x+" erroneo");  
    //... code to execute if no error...  
}
```



Constructor call

# Error Handling

## Exceptions: **catching**

- A **catch** instruction is similar to the definition of a one-parameter function which will be called if the corresponding exception is thrown:

### Catching

```
public static void main() {  
    try {  
        LlamameConCuidado(-0);  
    } catch (Exception ex) {  
        System.err.println("No tuviste cuidado: "+ ex);  
        ex.printStackTrace();  
    }  
}
```

# Error Handling

## Exceptions: **specification**

- Java and C++ provide syntax (and forces you to use that syntax) to allow you to politely tell the client programmer what exceptions a method throws (directly or indirectly), so the client programmer can handle them.
- This is the **exception specification** and it's part of the method declaration, appearing after the argument list:

`throws (<list-of-types>)`

- The exception specification uses an additional keyword, **throws**, followed by a list of all the potential exception types.

```
public int f() throws E1, E2 { ... }
```



# Error Handling

## Java standard exceptions

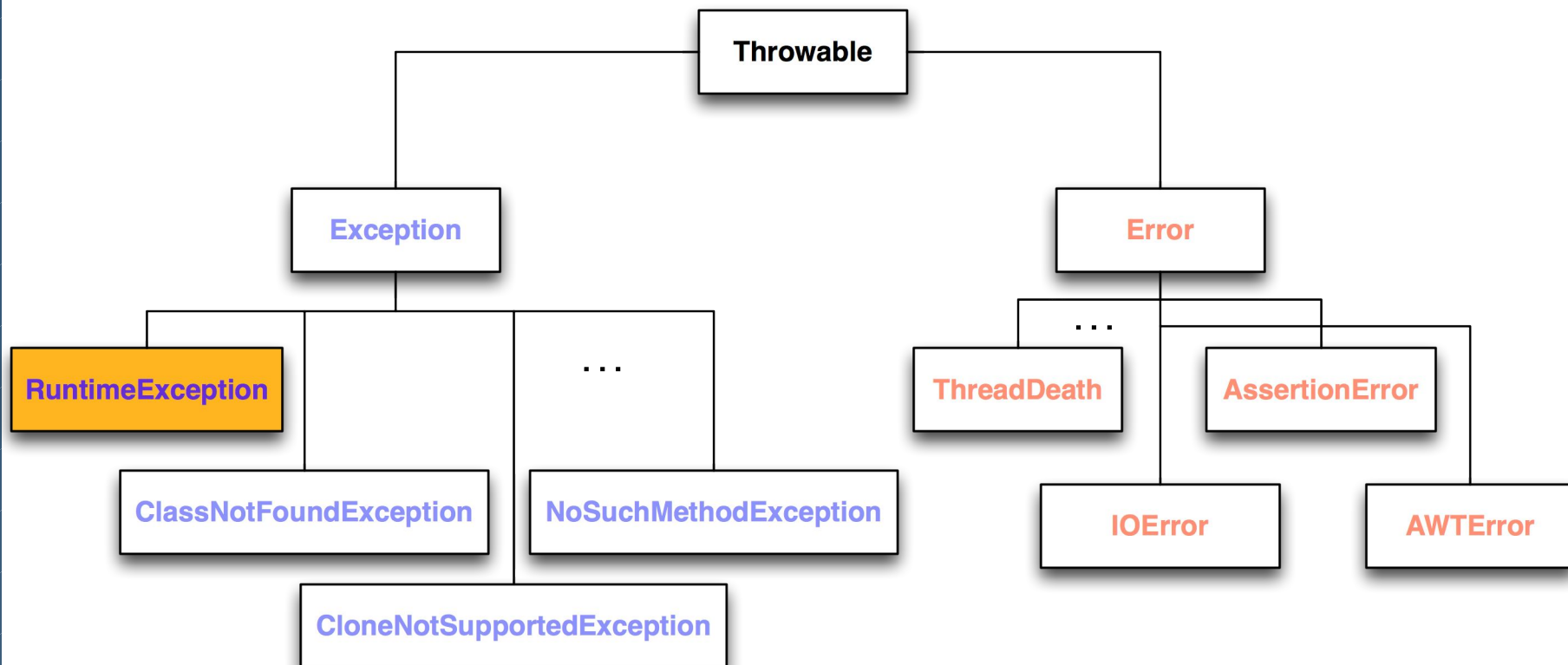
- The Java class **Throwable** (defined in `java.lang`) describes anything that can be thrown as an exception. All the exceptions thrown by the components of the Java API are types derived from `Throwable`, which contains, among others, the following methods:

```
class Throwable {  
    ...  
    public Throwable();  
    public Throwable(String message);  
    ...  
    public String getMessage();  
    public void printStackTrace();  
    public String toString(); // class name + message  
}
```

- All these methods are available when creating user exceptions.

# Error Handling

## Java standard exceptions



### Checked exceptions:

Exceptions that are checked and enforced at compile time. If a method throws any of these (except for *RuntimeException*), Java forces to include them in the exception specification.

# Error Handling

## Java standard exceptions

### **Exception:**

Execution errors in Java API or in our own classes.

### **Error:**

Used by the JVM to report system errors. Usually not caught.

### **RuntimeException:**

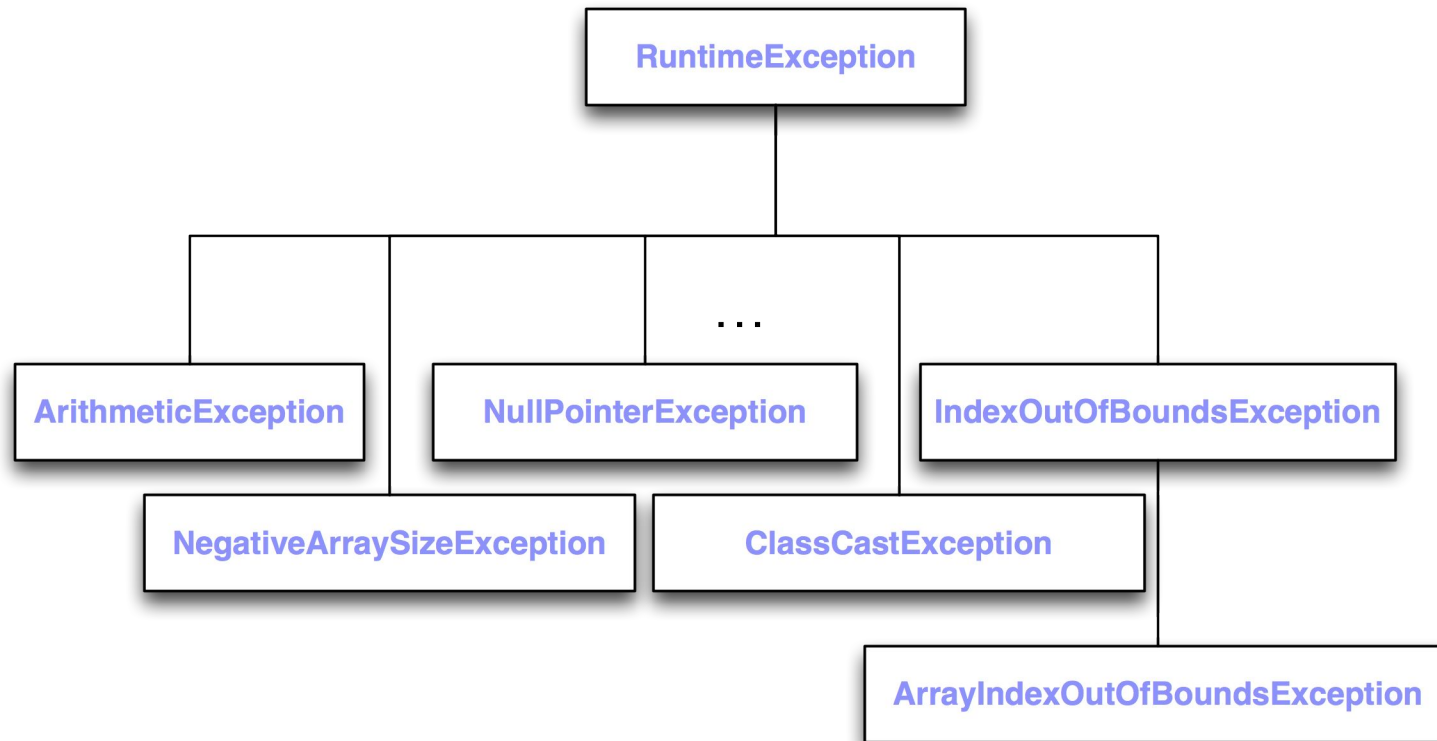
They indicate programming bugs, not user errors, so they are usually not caught as well.

All standard exceptions have at least two constructors: the default one and a second one with a String parameter describing the source of the error.

Any standard exception may be thrown, but those of types Error or RuntimeException are usually not.

# Error Handling

## RuntimeException: **unchecked exceptions**



### **Unchecked exceptions:**

As they indicate bugs, you don't usually catch a RuntimeException (or derivative classes): it's dealt with automatically. If you were forced to check for RuntimeExceptions, your code could get too messy. You never need to write an exception specification for them (e.g., for NullPointerException) because they are unchecked exceptions.

# Error Handling

## Exceptions: **specifications**

- An empty exception specification may imply that the method could throw any unchecked exception:

```
public int f() {  
    if (algo_falla)  
        throw new RuntimeException("Algo ha fallado");  
}
```

- Or no exception at all:

```
public int g() {  
    return 3+2;  
}
```

# Error Handling

## User exceptions

- The JDK exception hierarchy can't foresee all the errors you might want to report, so you can create your own.
- Advantages:
  - Extra information may be included in the exception.
  - Exceptions may be organized as a class hierarchy.
- To create your own exception class, you must inherit from an existing exception class, preferably one that is close in meaning to your new exception.

```
class miExcepcion extends Exception {  
    private int x;  
  
    public miExcepcion(int a, String m) {  
        super(msg); x=a; }  
    public String queHaPasado() { return getMessage(); }  
    public int getElCulpable() { return x; }  
}
```

# Error Handling

## Order of exception catching

- **The order of the catch clauses is important.**

```
try {  
    if (error1) throw new miExcepcion("ERROR 1");  
    If (error2) throw new RuntimeException("ERROR 2");  
    If (error3) throw new Exception("ERROR 3");  
}  
catch (miExcepcion e1) { ... handle error 1 ... }  
catch (RuntimeException e2) { ... handle error 2 ... }  
Catch (Exception e3) { ... handle error 3 ... }
```

If 'catch (Exception e3)' was first, all the errors would be treated as error3. The Java compiler detects this wrong situation: "Unreachable catch block for RuntimeException. It is already handled by the catch block for Exception"

# Error Handling

## try blocks nesting

- try/catch blocks may be nested

```
try {  
    if (error1) throw new miExcepcion("ERROR 1");  
    try {  
        if (error2) throw new RuntimeException("ERROR 2");  
        try {  
            if (error3) throw new Exception("ERROR 3");  
        }  
        catch (Exception e3) { ... handle error 3 ... }  
    }  
    catch (RuntimeException e2) { ... handle error 2 ... }  
}  
catch (miExcepcion e1) { ... handle error 1 ... }
```



# Error Handling

## Exceptions in constructors

- If an exception occurs inside of a constructor (and it is not caught in it), the object is not created.
- This usually happens when a value in one of the parameters does not allow to create a valid object.

Options:

- Throwing a user exception
- Throwing `IllegalArgumentException`

```
class Construcccion {  
    public Construcccion(int x) throws IllegalArgumentException {  
        if (x<0)  
            throw IllegalArgumentException("No admito negativas.");  
    }  
  
    public static void main(String args[]) {  
        try {  
            Construcccion c = new Construcccion(-3);  
        } catch (Exception ex) { ... 'c' has not been created ... }  
    }  
}
```

# Error Handling

## Rethrowing an exception

- Sometimes you will want to rethrow the exception that you just caught (particularly when you use `Exception` to catch any exception). Since you already have the reference to the current exception, you can simply rethrow that reference.
- Rethrowing an exception causes it to go to the exception handlers in the next-higher context, where handling of the error will be finished.
- Possibilities:
  1. Rethrow the same exception
  2. Throw a new exception

# Error Handling

## Rethrowing an exception

- First possibility: rethrow the same exception

```
public class Rethrowing {
    public static void f() throws Exception {
        throw new Exception("thrown from f()");
    }
    public static void g() throws Exception {
        try {
            f();
        } catch (Exception e) {
            // deal with part of the problem here...
            throw e; // rethrow the exception
        }
    }

    public static void main(String[] args) {
        try {
            g();
        } catch (Exception e) {
            // finish handling the issue generated in f()
        }
    }
}
```

# Error Handling

## Rethrowing an exception

- Second possibility: throw a new exception

```
class OneException extends Exception {}
class TwoException extends Exception {}

public class RethrowNew {
    public static void f() throws OneException {
        throw new OneException();
    }
    public static void g() throws TwoException {
        try {
            f();
        } catch(OneException e) {
            // deal with part of the problem here...
            throw new TwoException(); // throw a new exception
        }
    }

    public static void main(String[] args) {
        try {
            g();
        } catch(TwoException e) {
            // finish handling the issue generated in f()
        }
    }
}
```

# Error Handling

## Exceptions: summary

- **Advantages:**

- Code for error handling is separated from the normal code
- Different kind of errors are grouped under a meaningful hierarchy
- Client code is forced to specifically deal with (or ignore) error conditions

- **Disadvantages:**

- The system for handling exception flow is overloaded.

# Error Handling Reminders

- If an exception occurs:
  - Is some cleanup necessary?
  - Should it be caught at the same scope where it is thrown?
  - Should it be passed to the next higher level?
  - Should it be processed partially and rethrown (same exception or a different one) to the next higher level?
- The usual practice is to delegate exception handling to the caller:
  - “Do not catch an exception if you do not know what to do with it”

# Error Handling

## Example

- TASK

- **Define a user exception called** `ExcepcionDividirPorCero` to be thrown by the next method upon trying to divide by zero:

```
class Division {  
    static float div(float x, float y)  
    { return x/y; }  
}
```

- Write a program which invokes `div()` and handles the exception correctly.

# Error Handling

## Example

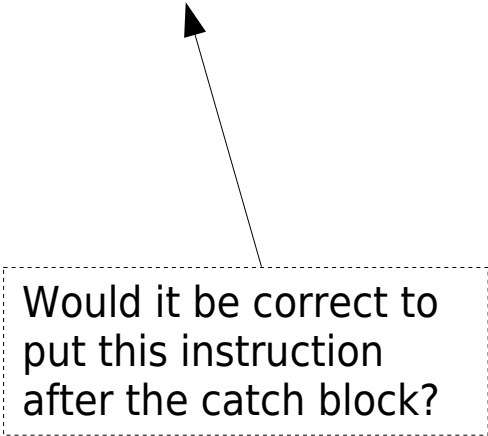
- SOLUTION:

```
class ExcepcionDividirPorCero extends Exception {  
    public ExcepcionDividirPorCero(String msg)  
    { super(msg); }  
}
```



# Error Handling Example

```
class Divisora {  
  
    public static void main(String args[])  
    {  
        double dividendo, divisor, resultado;  
        dividendo = 4.0;  
        divisor = 0.0;  
        try {  
            resultado = Division.div(dividendo, divisor);  
            System.out.println(dividendo+"/"+divisor+"="+resultado);  
        }  
        catch (ExcepcionDividirPorCero exce)  
        {  
            System.err.println(exce.getMessage());  
            exce.printStackTrace();  
        }  
    }  
}
```



# Error Handling

## Exceptions: homework

### ■ Task

- Define a class `IntStack` which handles an integer stack. The stack will be created with a maximum capacity. An overflow situation in method `push()` has to rise a user exception `ExcepcionDesbordamiento`.
- Define class `IntStack`, its constructor and the method `push()`.
- Then, implement a test main program which continuously pushes elements in the stack, catches the exception when produced, and then aborts the execution of the program.

# Error Handling

## Bibliography

- Bruce Eckel. ***Piensa en Java 4ª edicion***
  - Ch. 12
- Bruce Eckel. ***Thinking in Java, 3rd. edition***
  - Ch. 9
- C. Cachero et al. ***Introducción a la programación orientada a objetos***
  - Ch. 5 (examples in C++)