Exception Handling

Look at this Problem

```java
public class Test {
     public static void main(String[] args) {
          System.out.println( 3/0 );
     }
}
```

The program terminates abnormally

```
>java Test
Exception in thread "main" java.lang.ArithmeticException:
     / by zero at Test.main(Test.java:3)
```

Examples of Abnormal Conditions

- The file you try to open does not exist
- Operands being manipulated are out of prescribed ranges
- Divided by 0
- Invalid input
- Network hangs up
- …

There are good reasons to perform error processing

- Notify the user of an error
- Save all work
- Allow users to gracefully exit the program

Traditional Error-handling

- Traditional error-handling methods before OO programming
  - Return a status code to indicate either success or failure
  - Assign an error code to a global variable
    - Other functions can examine
  - Terminate the program

In OO programming, they are unacceptable.

Exception Handling

- Exception handling in OO programming
  - A method of a server class detects an error then notifies the client class of it
    - A method of the server class *throws* an **`Exception`** object containing error information
  - The client class decides what to do about the error
    - The client class *catches* the **`Exception`** object and takes actions (**handles** the exception)

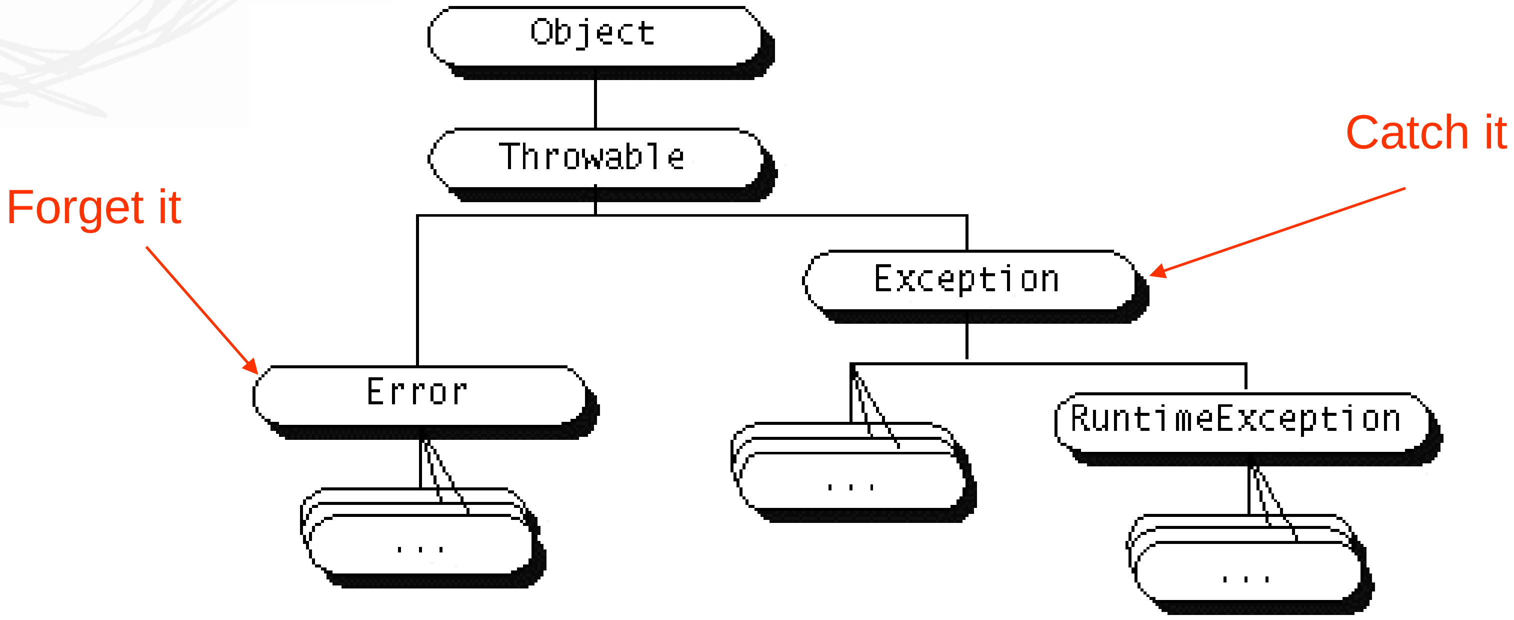
The **`Exception`** object containing error information is passed from where the error occurs (server class) to where the problem is handled (client class)

Java Exceptions

- Conditions that can occur in a correct program are ***checked exceptions***
  - Represented by the `Exception` class and listed by API
  - Typically a user error or a problem that cannot be foreseen by the programmer
    - e.g. File not found
  - *Cannot be ignored at the compilation time*
    - You must catch checked exceptions in your code
- Severe problems that are treated as fatal or situations that probably reflect program bugs are ***unchecked exceptions***
  - *Can be ignored at the compilation time*
    - If an exception is not caught in your own code, the system will catch it and terminate your program (as a penalty?)
  - ***Runtime exceptions***
    - Problem bug are represented by the `RuntimeException` class
    - Can be avoided by the programmer
      - e.g. Exceeding the end of an array
  - **Errors**
    - Fatal situations are represented by the `Error` class
    - Problems beyond the control of users or programmers
      - e.g. Running out of memory; exceeding the end of an array

Exception Class Hierarchy

An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.



Exception Handling

When runtime error occurs, Java class throws an exception

```
public class Test {
      public static void main(String[] args) {
            try {
                System.out.println( 3/0 );
             }
             catch (Exception ex) {
                System.out.println("Error: "+ex.getMessage());
            }
            System.out.println("Continues!");
      }
}
```

The program continues normally

```
>java Test
Error: / by zero
Continues!
```

Catching Multiple Exceptions

Look at API doc

As many as you need

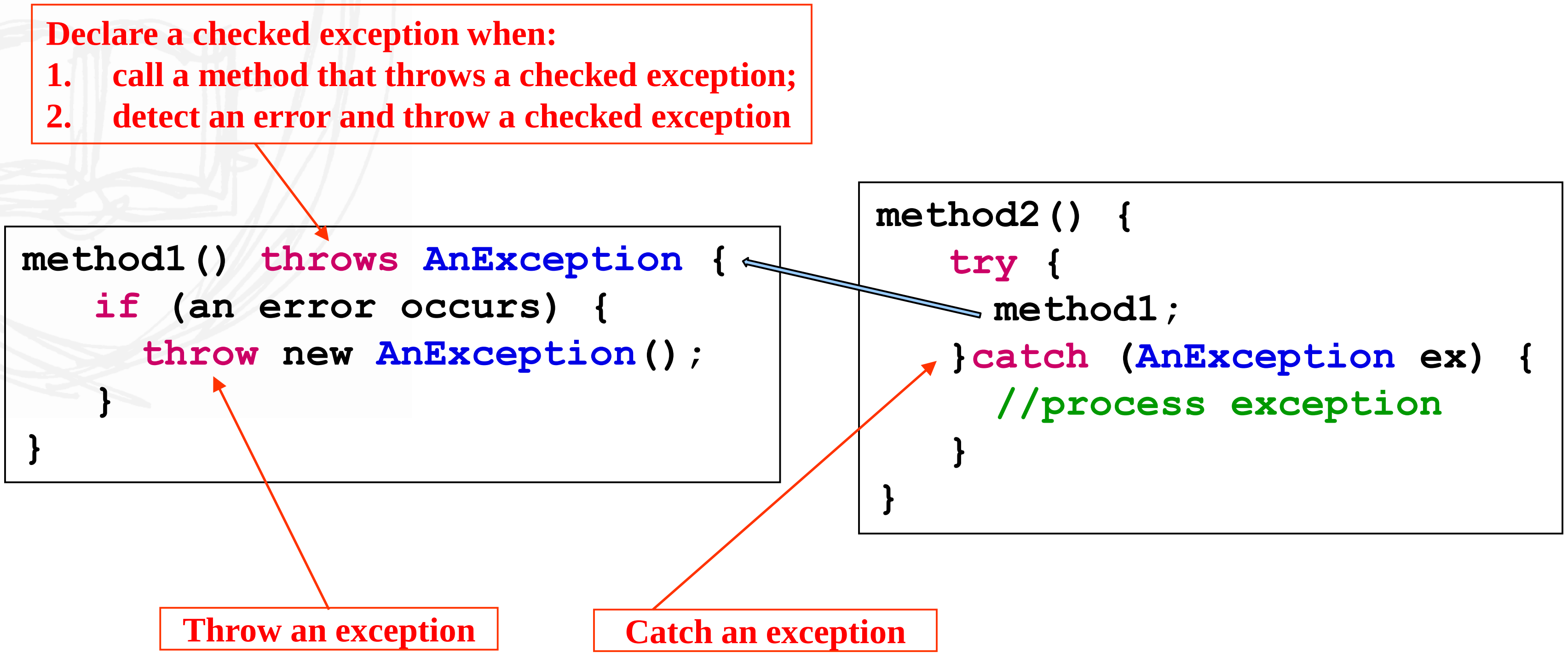
optional

```
try {
  // code that might throw exceptions;
  // if an exception is thrown, the program
  //    1. skips the remainder of the code;
  //    2. executes the handler code inside the catch clause
  // any code afterward would not be reached if error occurs

}
catch (AKindOfException ex1) {
    //exception-handling
    //Catch derived exceptions earlier
}
catch (AnotherKindOfException ex2) {
    //exception-handling
}
finally { //this clause executes, regardless of try/catch

     //do something here, eg. release resource
}
```

Catching Multiple Exceptions

```
try {
  // code that might throw exceptions;
}
catch (AKindOfException | AnotherKindOfException ex1)
{
    //Combined catching in SE7
}
catch (OtherKindOfException ex2) {
    //Catch some more
}
finally {
     //do something here, eg. release resource
}
```

Checked Exceptions



Declare a checked exception when:
1. call a method that throws a checked exception;
2. detect an error and throw a checked exception

```
method1() throws AnException {
   if (an error occurs) {
     throw new AnException();
   }
}
```

```
method2() {
   try {
     method1;
   }catch (AnException ex) {
     //process exception
   }
}
```

Throw an exception

Catch an exception

Unchecked exceptions:
1. Subclasses of **`Error`**
2. Subclasses of **`RuntimeException`**

Rethrowing Exceptions

```
try {
    statements;
}catch(TheException e) {

  //perform operations before exits;

  throw e;
}
```

Rethrowing the exception so that other handlers get a chance to process the exception

Define Your Own Exceptions

```
class YourException extends Exception {

    public YourException() {}
    public YourException(String message){
        super(message);
        //There can be more code here,
        //but often there is none.
    }
}
```

```
throw new YourException();
```

or

```
throw new YourException(“My exception thrown.”);
```

Try-with-Resources

```
try (Resource res = … )
{
   // work with res
}

// The resource must belong to a class that implements the
AutoClosable interface

// No matter how the try block exits, the res.close() is called
automatically

// This avoids two nested try/finally statements
```

```
try ( Scanner in = new Scanner(new FileInputStream(“myFile.txt”)),
      PrintWriter out = new PrintWriter(“MyOutput.txt”))
{
  while (in.hasNext())
     out.println(in.next().toUpperCase());
}
```

Common Exceptions

- `ArithmeticException`
- `NullPointerException`
- `NegativeArraySizeException`
- `ArrayIndexOutOfBoundException`
- `SecurityException`

Notes on Exception Handling

- Exception handling usually requires more time and resources
- Exception handling should not be used to replace simple test with ***`if`*** statements
- Use exception handling for common exceptions in multiple classes; use ***`if`*** statement to handle simple errors in individual classes
- An error message appears on the console with unhandled exceptions, but the GUI application may or may not continue running

Robust Program Design

When an exception is caught

- Print a message and terminate
- Log the error and resume
- Fix the error and resume

Assertions

- Used to test invariants

```
    assert condition

    assert condition : expression
```

- If **`condition`** evaluates false, an **`AssertionError`** is thrown

- The second argument is converted to a string and used as descriptive text in the **`AssertionError`** message

Example

```
y = Math.sqrt(x);

// double check to ensure the x is not negative
if  (x<0) throw new IllegalArgumentException(“x<0”);
```

Staying in the code

Use assertion

Enabled at runtime

```
assert x >= 0;

or

assert x >=0 : x;
```

Assertion Enabling and Disabling

- By default, assertions are ***disabled***

```
java –enableassertion MyApp
java –ea MyApp


java –disableassertion MyApp
java –da MyApp
```

Recommended Uses of Assertions

- Locating internal errors during testing
- Documentation and verification of assumptions and internal logic in a single method
  - Internal invariants
  - Control flow invariants
  - Postconditions and class invariants
- Not for precondition checking on public methods
- Assertion checks are turned on only during development and testing

Three Mechanism to Deal with System Failures

- Throwing an exception
- Using Assertion
- Logging

Logging

- Use logging to gain insight into program behaviour
  - Used many `System.out.println()` calls?
- Logging can be suppressed
- Log records can be directed to different handlers
  - Console, a file, a stream, memory or a TCP socket.
- Log records can be filtered and formatted

Basic Logging

```
Logger.getGlobal().setLevel(Level.ALL);
Logger.getGlobal().info("This is for information only");
Logger.getGlobal().warning("This is a warning");
Logger.getGlobal().severe("This is a severe failure");
```

The record is printed as:

class name method name

```
Aug 1, 2012 9:35:23 PM TestLogger main
INFO: This is for information only
Aug 1, 2012 9:35:23 PM TestLogger main
WARNING: This is a warning
Aug 1, 2012 9:35:23 PM TestLogger main
SEVERE: This is a severe failure
```