CMPS 251

Read Chapter 11

Exception Handling

Dr. Abdelkarim Erradi

CSE@QU

Outline

- Exception handling using **try-catch-finally**
- Exception Types
- Throwing an Exception
- Custom Exceptions

try-catch-finally block

Back

Definition

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of execution.
- Examples
  - A program is going to read a file, but *the file is missing*
  - A program is reading an array, but the *out of bound* case occurs
  - A program is receiving a file, but the network connection fails
- With exception handling, a program can continue executing (rather than terminating) after dealing with the exception.
  - Yields robust and fault-tolerant programs (i.e., programs that can deal with problems as they arise and continue executing).

Example of throwing an exception

```
// Fig. 8.1: Time1.java
// Time1 class declaration maintains the time in 24-hour format.

public class Time1
{
   private int hour; // 0 - 23
   private int minute; // 0 - 59
   private int second; // 0 - 59

   // set a new time value using universal time; throw an
   // exception if the hour, minute or second is invalid
   public void setTime( int h, int m, int s )
   {
      // validate hour, minute and second
      if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
         ( s >= 0 && s < 60 ) )
      {
         hour = h;
         minute = m;
         second = s;
      } // end if
      else
         throw new IllegalArgumentException(
            "hour, minute and/or second was out of range" );
   } // end method setTime
```

throws an IllegalArgumentException if the data validation fails in set method

Example of catching and handling the Exception

```

      // attempt to set time with invalid values
      try
      {
         time.setTime( 99, 99, 99 ); // all values out of range
      } // end try
      catch ( IllegalArgumentException e )
      {
         System.out.printf( "Exception: %s\n\n", e.getMessage() );
      } // end catch
```

Catch and handle the Exception

Method setTime() and Exception Handling

- For incorrect values, `setTime` throws an exception of type `IllegalArgumentException` (lines 23–24)
  - Notifies the client code that an invalid argument was passed to the method.
  - The **`throw`** statement (line 23) creates a new object of type `IllegalArgumentException`. In this case, we call the constructor that allows us to specify a custom error message.
  - After the exception object is created, the **`throw`** statement immediately terminates method `setTime()` and the exception is returned to the client code that attempted to set the time.
  - The client can use `try`…`catch` to catch exceptions and attempt to recover from them.

Some common exceptions …

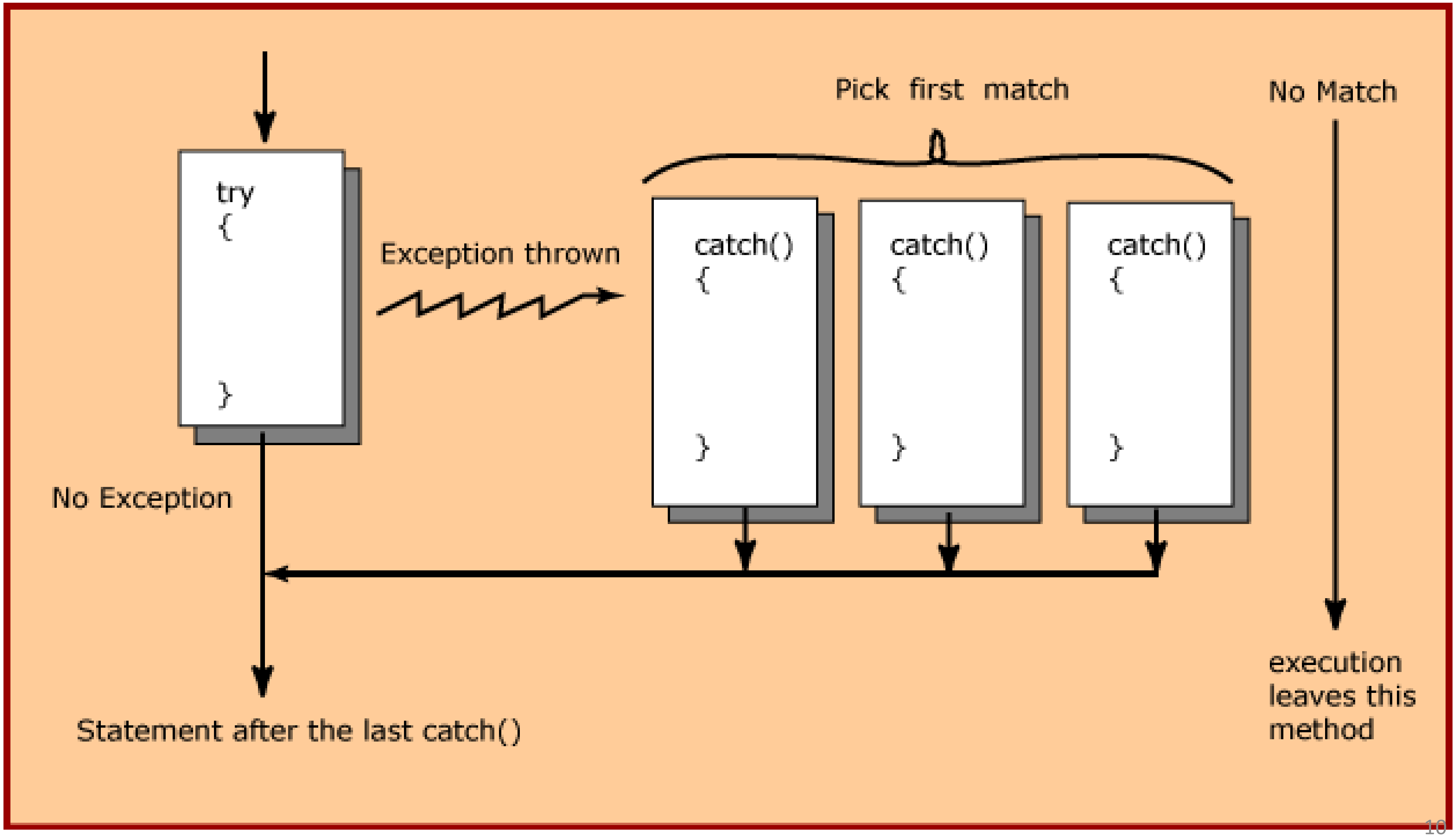
- `ArrayIndexOutOfBoundsException` occurs when an attempt is made to access an element past either end of an array
- A `NullPointerException` occurs when a `null` reference is used where an object is expected
- `ClassCastException` occurs when an attempt is made to cast an object that does not have an *is-a* relationship with the type specified in the cast operator.
- `IOException` may occur when reading or writing to files

For a list of subclasses of Exception, see
`http://download.oracle.com/javase/7/docs/api/java/lang/Exception.html`

Exception Handling using try-catch-finally

```
try {
   //...something might have exception
}
catch (SomeExceptionClass e) {
   //handle the exception here
}
finally {
   //release resources
}
```

How try and catch Work



How try and catch Work (1 of 2)

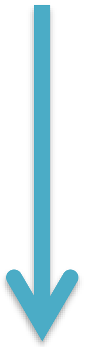
- If an exception occurs in a `try` block, the `try` block terminates immediately and program control transfers to the first `catch` block whose type matches the type of the exception that occurred.
  - If there are remaining statements after the statement that causes the exception, those remaining statements won't be executed.
- After the exception is handled, any remaining `catch` blocks are ignored, and execution resumes at:
  - The `finally` block, if one is present
  - Or at the first line of code after the `try`…`catch` `sequence`
  - **Control does <u>not</u> return to the try block.**

How try and catch Work (2 of 2)

- If no catch{} block matches the exception, the **execution leaves this method**
  - The unhandled exception is passed to the caller
- If no exception occurs in the `try` block, the `catch` blocks are skipped and control continues with the first statement after the `catch` blocks
  - But, the `finally` block , if one is present, will execute whether or not an exception occurs in the corresponding `try` block.

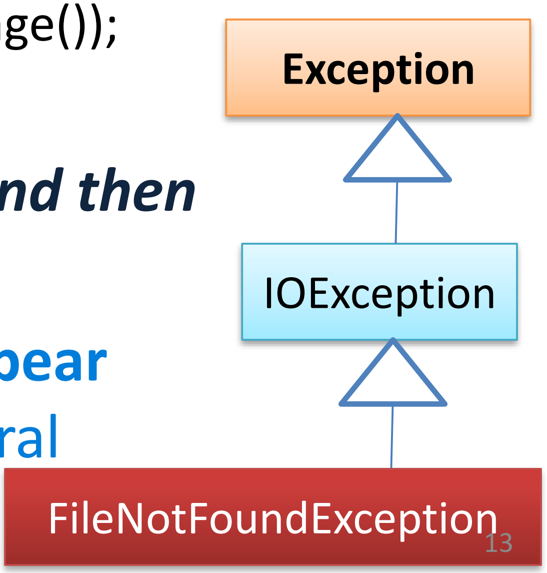
Try with multiple catch blocks

```
try {
    //maybe read a file or something…
}

catch (FileNotFoundException e) {
   System.out.println("FileNotFoundException: "  + e.getMessage());
}

catch (IOException e) {
   System.out.println("Caught IOException: " + e.getMessage());
}
```



- ***Why we catch FileNotFoundException first, and then IOException?***

=> The **most specific exception types should appear first** in the structure, followed by the more general exception types.



Multi-Catch

```
try {
...
} catch (ClassCastException e) {
   doSomethingClever(e);
   throw e;
} catch(InstantiationException |
     NoSuchMethodException |
     InvocationTargetException e) {
   // Useful if you do generic actions
   log(e);
   throw e;
}
```

Log the exception then rethrow it

try-with-resources

- The **try-with-resources** is a try statement that declares one or more resources. A *resource* is an object that must be closed after the try block.
  - Any object that implements **java.lang.AutoCloseable** can be used as a resource.
  - "**try-with-resources**" will auto-close the resource (e.g., an open file) that was created in the try

```
try (Scanner inputFile = new Scanner(new FileInputStream(filePath))) {
    String line;
    while (inputFile.hasNext()) {
        line = inputFile.nextLine();
        fileLines.add(line);
    }
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

inputFile will be closed automatically after try block. No need to add a finally block

Superclass/Subclass Exceptions

- A `catch` parameter of an exception can also catch **all** of that exception type's subclass types.
  - Enables `catch` to handle related exceptions with a concise notation
  - Allows for polymorphic processing of related exceptions
- Catching related exceptions in one `catch` block makes sense only if the handling behavior is the same for **all** subclasses.
- You can also catch each subclass type individually if those exceptions require different processing.
- If multiple `catch` blocks match a particular exception type, only the first matching `catch` block executes.

The `finally` block

- The finally block is used to **release resources** acquired in the try block such as closing files, database connections and network connections .
  - Programs that obtain resources must close them properly to prevent resource leaks and make them available for use in other programs.
- The `finally` block will ***always*** execute whether or not an exception occurs in the corresponding `try` block.
- The `finally` block will execute if a `try` block exits by using a `return`, `break` or `continue` statement or simply by reaching its closing brace.
- The `finally` block will *not* execute if the application terminates immediately by calling method System.exit().
- Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated => it allows the programmer to **avoid having cleanup code accidentally bypassed** by a return, continue, or break.

The finally clause

```
try {
    Protect one or more statements here.
}
catch(Exception e) {
    Report and recover from the exception here.
}
finally {
    Perform any actions here common to whether or
        not an exception is thrown.
}
```

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose no exceptions in the statements

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed

Trace a Program Execution

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Next statement in the method is executed

Trace a Program Execution

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
finally {
   finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

Trace a Program Execution

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
finally {
   finalStatements;
}

Next statement;
```

The exception is handled.

Trace a Program Execution

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
finally {
   finalStatements;
}

Next statement;
```

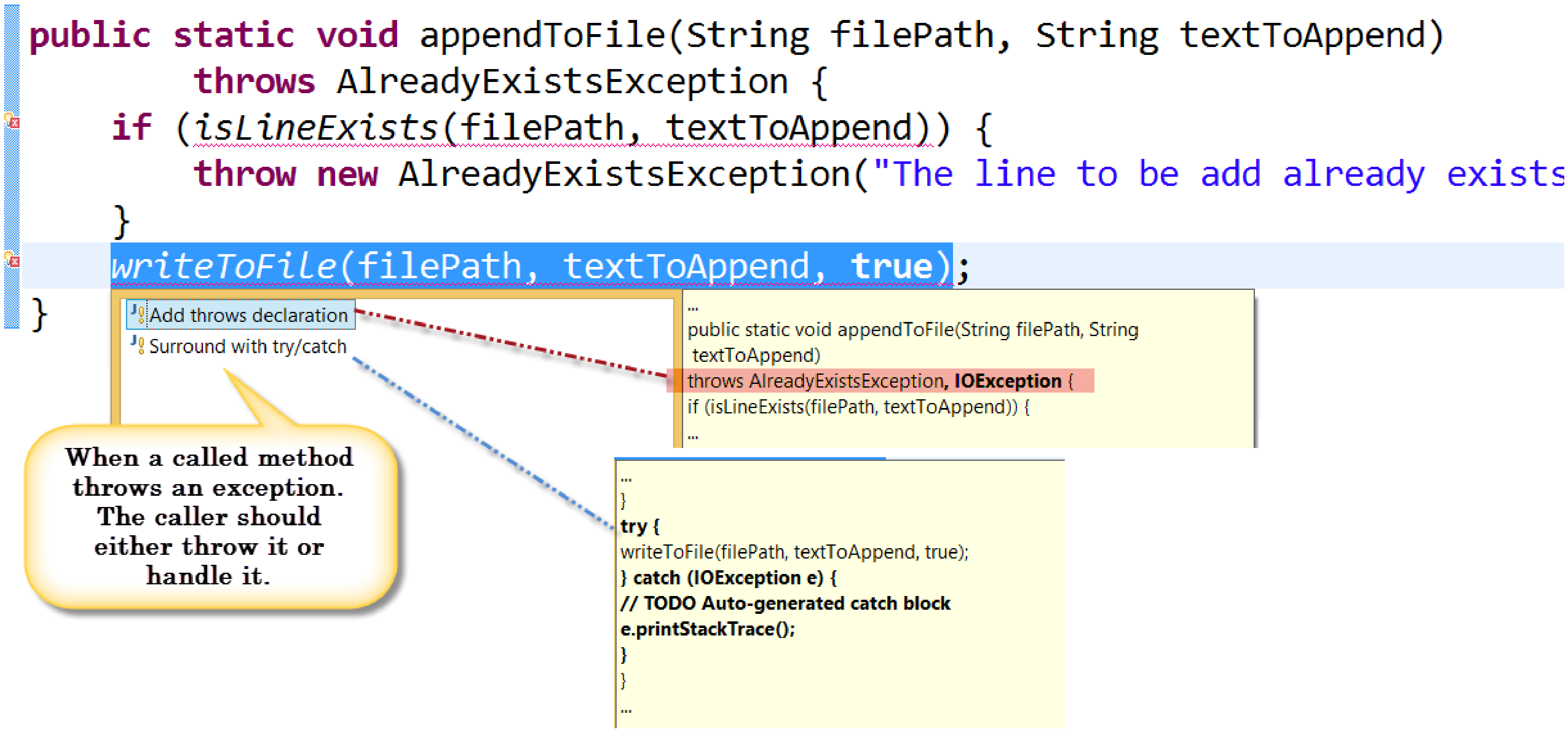
The final block is always executed.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

When a called method throws an exception. The caller should either throw it or handle it.



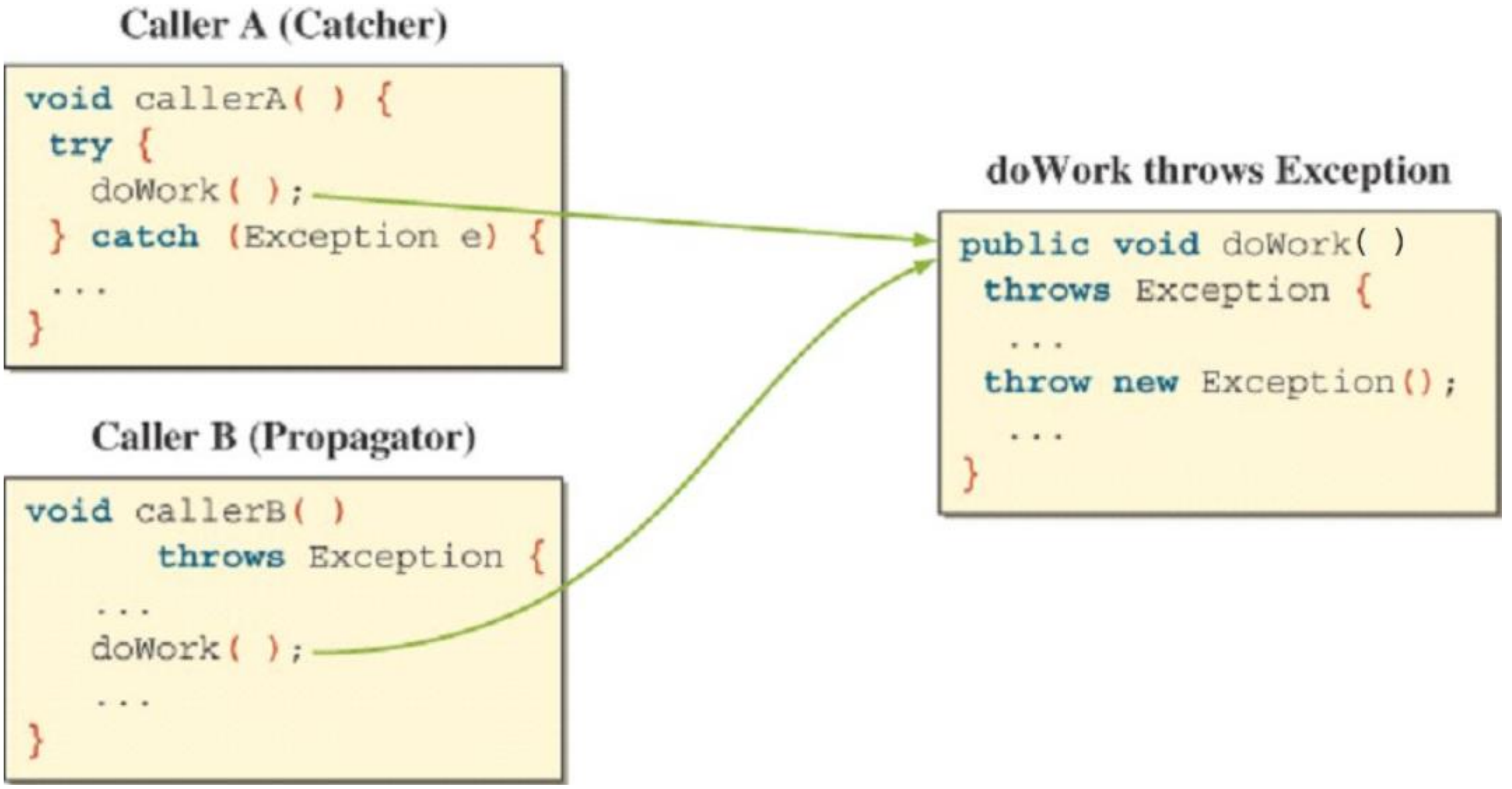
```
public static void appendToFile(String filePath, String textToAppend)
        throws AlreadyExistsException {
    if (isLineExists(filePath, textToAppend)) {
        throw new AlreadyExistsException("The line to be add already exists
    }
    writeToFile(filePath, textToAppend, true);
}
```

```
...
public static void appendToFile(String filePath, String
 textToAppend)
throws AlreadyExistsException, IOException {
if (isLineExists(filePath, textToAppend)) {
...
```

```
...
}
try {
writeToFile(filePath, textToAppend, true);
} catch (IOException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}
...
```

When a called method throws an exception. The caller should either throw it or handle it.

When an exception is throws by a method, the method caller could be either: Catcher or Propagator


Caller A (Catcher)

```
void callerA( ) {
 try {
   doWork( );
 } catch (Exception e) {
 ...
}
```

Caller B (Propagator)

```
void callerB( )
      throws Exception {
   ...
   doWork( );
   ...
}
```

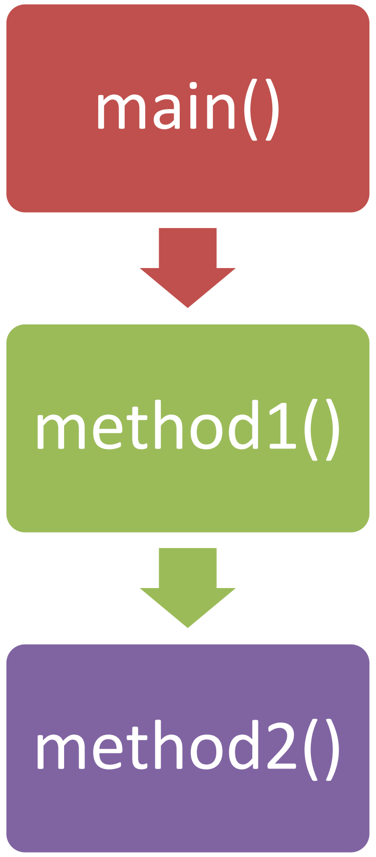
doWork throws Exception

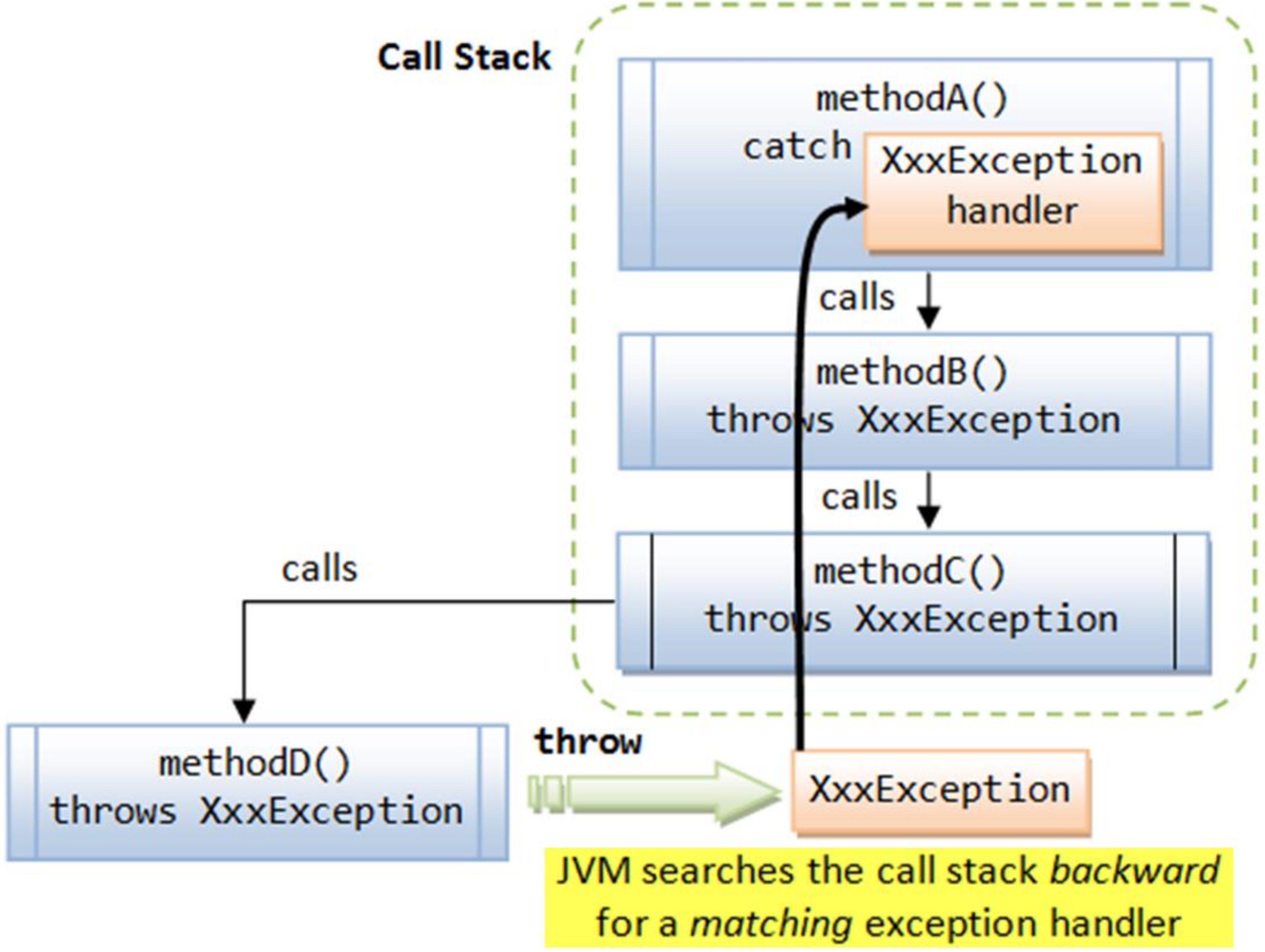
```
public void doWork( )
 throws Exception {
   ...
 throw new Exception();
   ...
}
```

Uncaught exceptions

- Uncaught exceptions **propagate up** the **Call Stack**.
- Uncaught exceptions that are still not handled in the ***main*** cause JVM default exception handler to run. This displays an error message and a complete **execution stack trace** then the application terminates.
  - The execution stack trace shows the tracing of the location where errors occur.

Call Stack

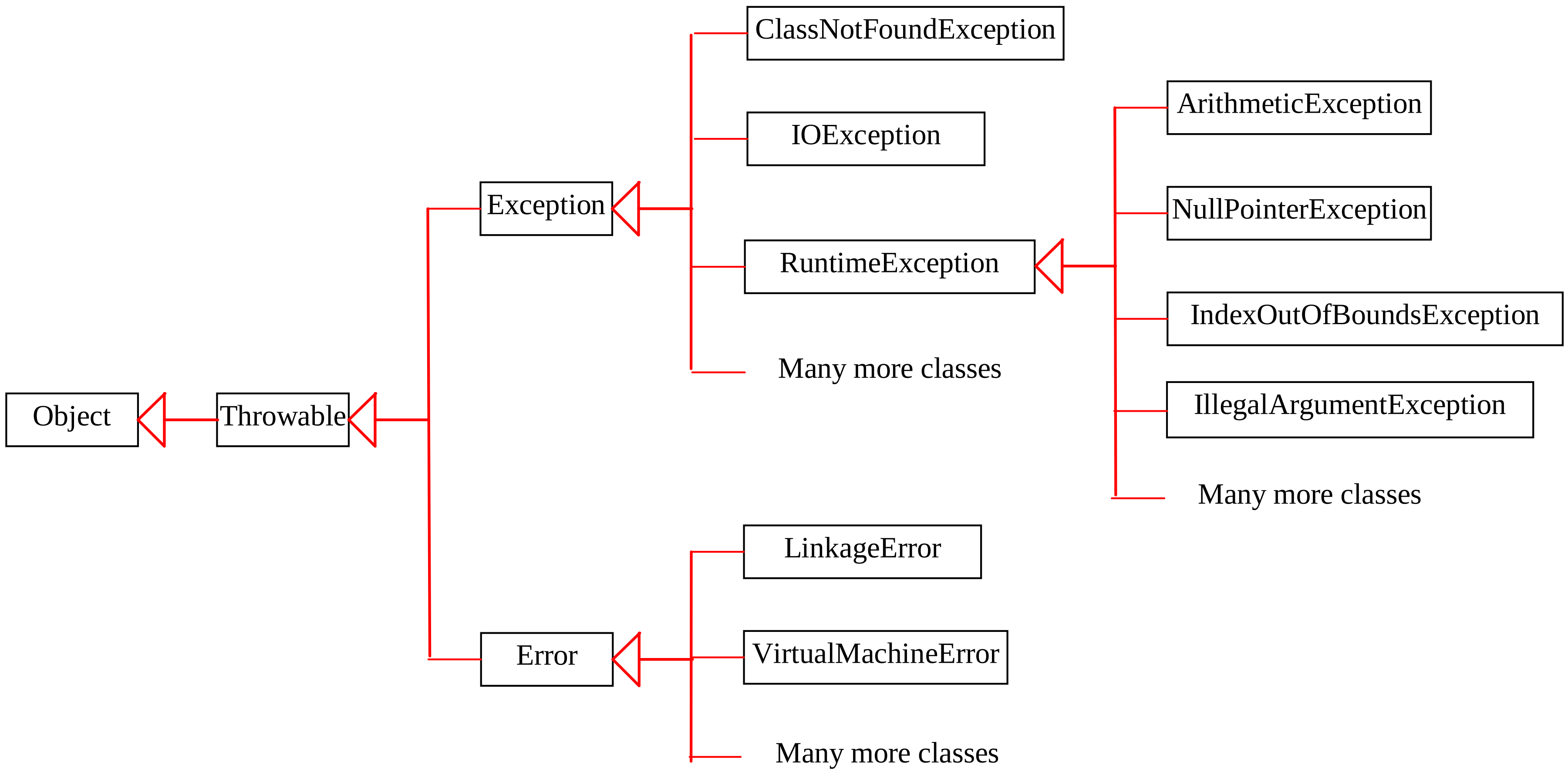



Call Stack
methodA()
catch
XxxException handler
calls
methodB() throws XxxException
calls
methodC() throws XxxException
calls
methodD() throws XxxException
throw
XxxException
JVM searches the call stack *backward* for a *matching* exception handler

Exception Types

Back

Exception Types



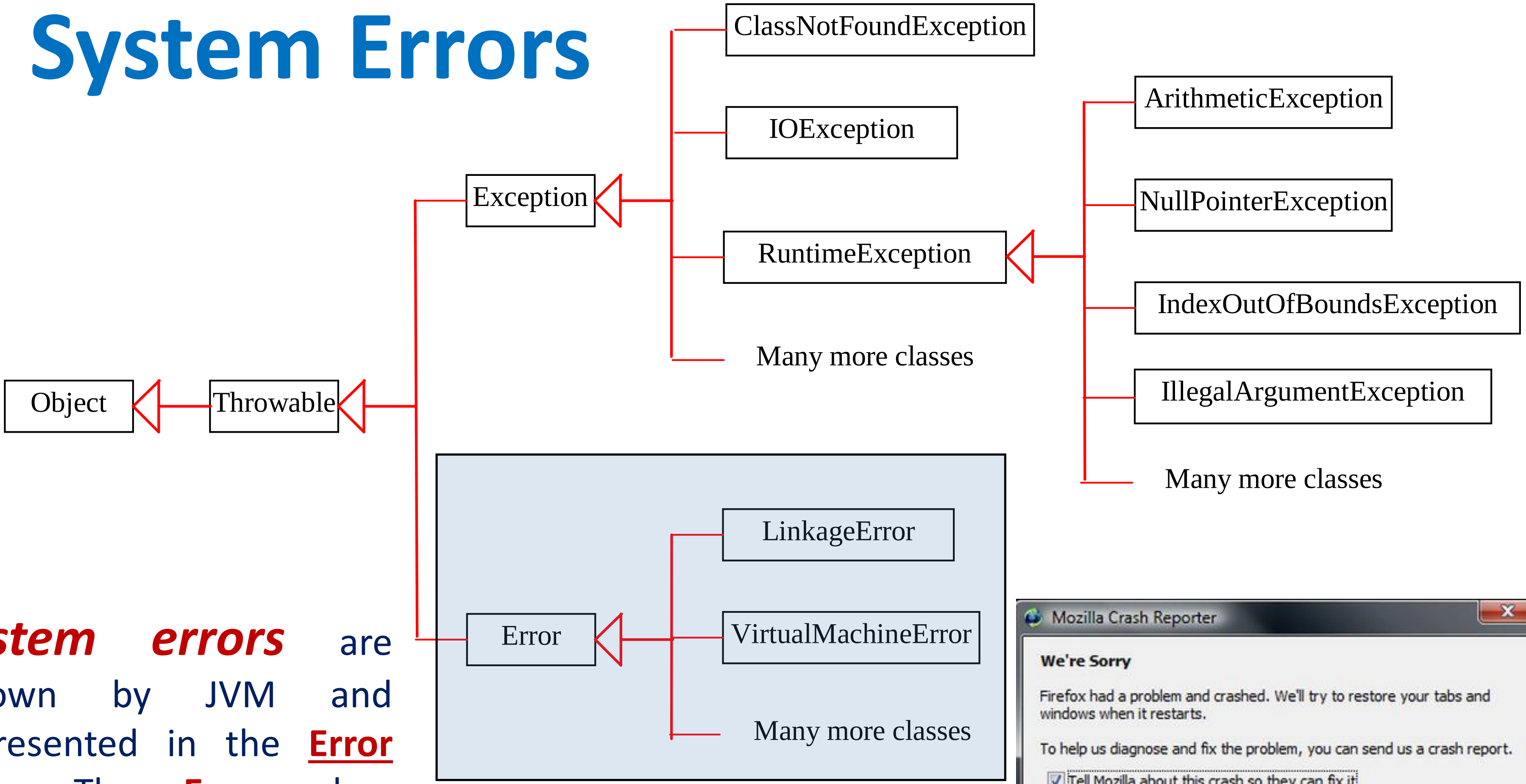
Java Exception Hierarchy

- Exception classes inherit directly or indirectly from class Exception, forming an inheritance hierarchy.
  - Can extend this hierarchy with your own exception classes.
- The previous slide shows a small portion of the inheritance hierarchy for class Throwable, which is the superclass of class `Exception`.
  - Only `Throwable` objects can be used with the exception-handling mechanism.
- Class `Throwable` has two subclasses: `Exception` and `Error`
  - An Error indicates a serious system problem that cannot be recovered, e.g., JVM crashes.
  - An Exception indicates that a problem occurred, but it is not a serious system problem.
  - Most programs you write will throw and catch Exceptions as opposed to Errors.

Three kinds of exceptions

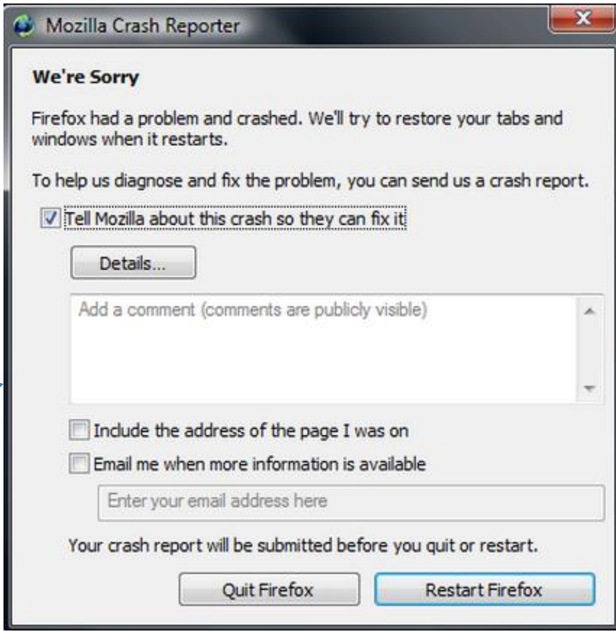
- ***Checked exception*** *(Java forces you to handle them)*
  - These are exceptional conditions that a well-written application should anticipate and recover from.
  - They occur usually interacting with outside resources/ network resources e.g. database problems, network connection errors, missing files etc.
  - E.g., FilNotFoundException
- ***Unchecked exceptions*** *(Cannot recover from them)*
  - Error
    - These are exceptional conditions that are external to the application and outside its control. The application usually cannot anticipate or recover from them. e.g., **Out of memory exception**
  - Runtime exception
    - These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from.
    - These usually indicate **programming bugs**, such as logic errors or improper use of an API (e.g., **NullPointerException**).

System Errors



System errors are thrown by JVM and represented in the **Error** class. The **Error** class describes internal system errors. Such errors rarely occur. If one does, **there is little you can do beyond notifying the user** and trying to terminate the program gracefully.

Example of how Firefox handles errors causing crash


Mozilla Crash Reporter

We're Sorry

Firefox had a problem and crashed. We'll try to restore your tabs and windows when it restarts.

To help us diagnose and fix the problem, you can send us a crash report.

[✓] Tell Mozilla about this crash so they can fix it

Details...
Add a comment (comments are publicly visible)

[ ] Include the address of the page I was on

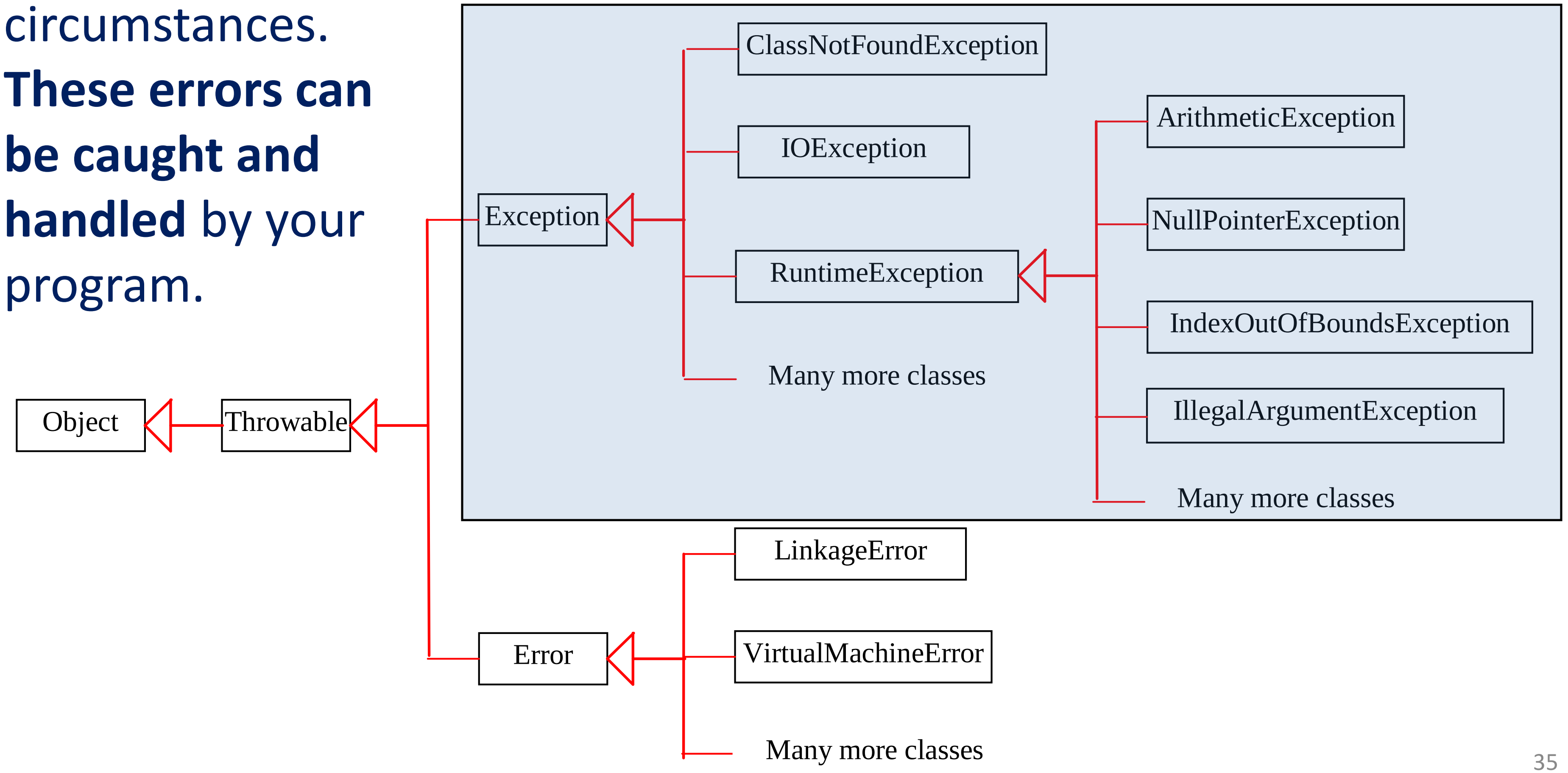
[ ] Email me when more information is available

Enter your email address here

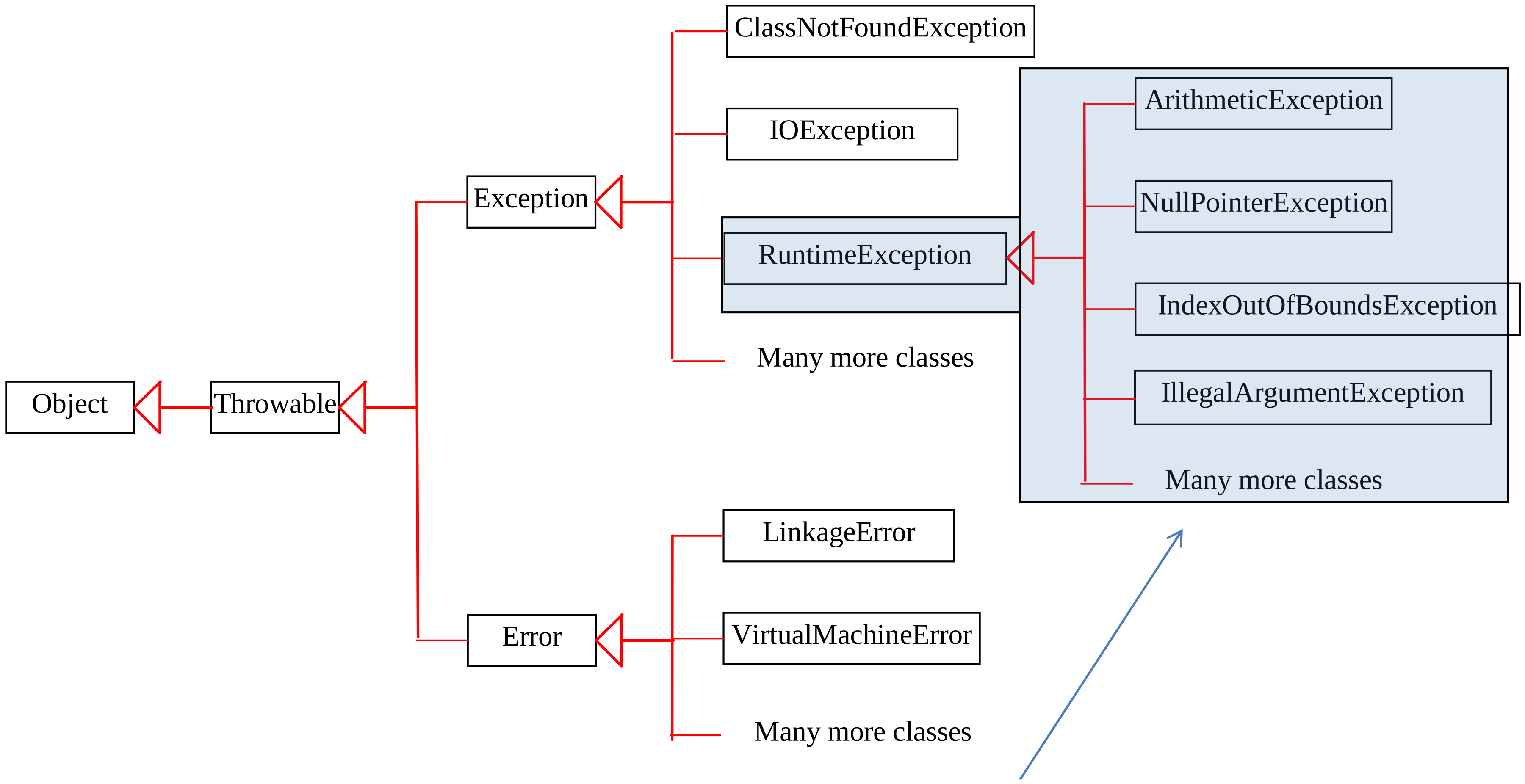
Your crash report will be submitted before you quit or restart.

Exceptions

Exception describes errors caused by your program and external circumstances. **These errors can be caught and handled** by your program.



Runtime Exceptions

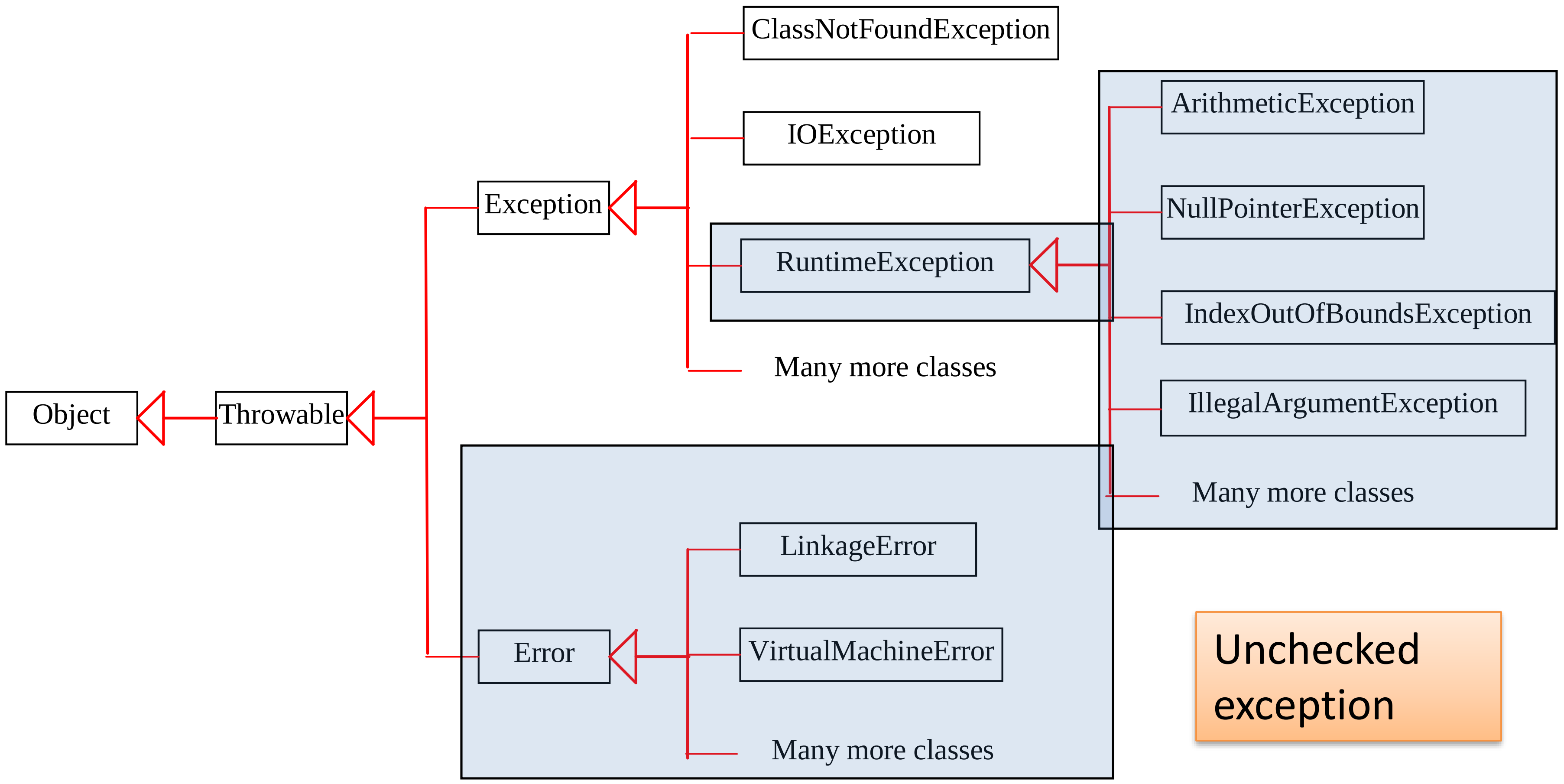


RuntimeException is caused by programming errors, such as accessing a null object, bad casting, accessing an out-of-bounds array, and arithmetic errors.

Checked Exceptions vs. Unchecked Exceptions

- ***RuntimeException***, ***Error*** and their subclasses are known as ***unchecked exceptions***.
  - They are often logic errors that are not recoverable
  - For example, a NullPointerException is thrown if you access a null object; an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array.
  - Java does not mandate you to **write code to catch unchecked exceptions**.
- All other exceptions are known as ***checked exceptions***, meaning that the compiler forces the programmer to check and deal with the exceptions.

Unchecked Exceptions



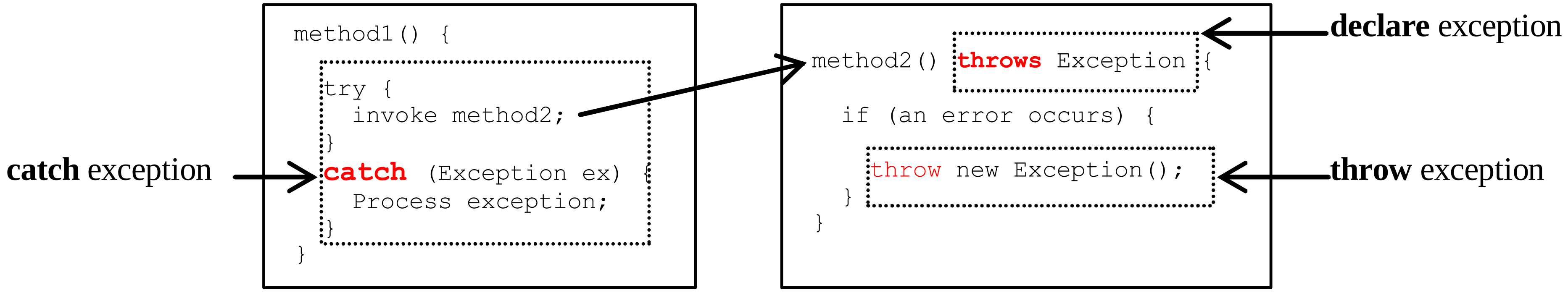
Unchecked exception

Throwing an Exception

< Back

Declaring, Throwing, and Catching Exceptions

- Use the try block to delimit the code in which exceptions might occur
- Throw exceptions to indicate a problem
- Use catch blocks to specify exception handlers



```
method1() {

  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

The throws clause

- Specifies the exceptions a method may throw.
- Appears after the method's parameter list and before the method's body. E.g., `public void myMethod()   throws IOException`
- Contains a comma-separated list of the exceptions that the method will throw if various problems occur.
  - May be thrown by statements in the method's body or by methods called from the body.
- Method can throw exceptions listed in its `throws` clause or their subclasses.
  - e.g., NullPointerException is a subclass of RuntimeException.
- Clients of a method with a `throws` clause are thus informed that the method may throw exceptions.

Throwing Exceptions

- When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example:

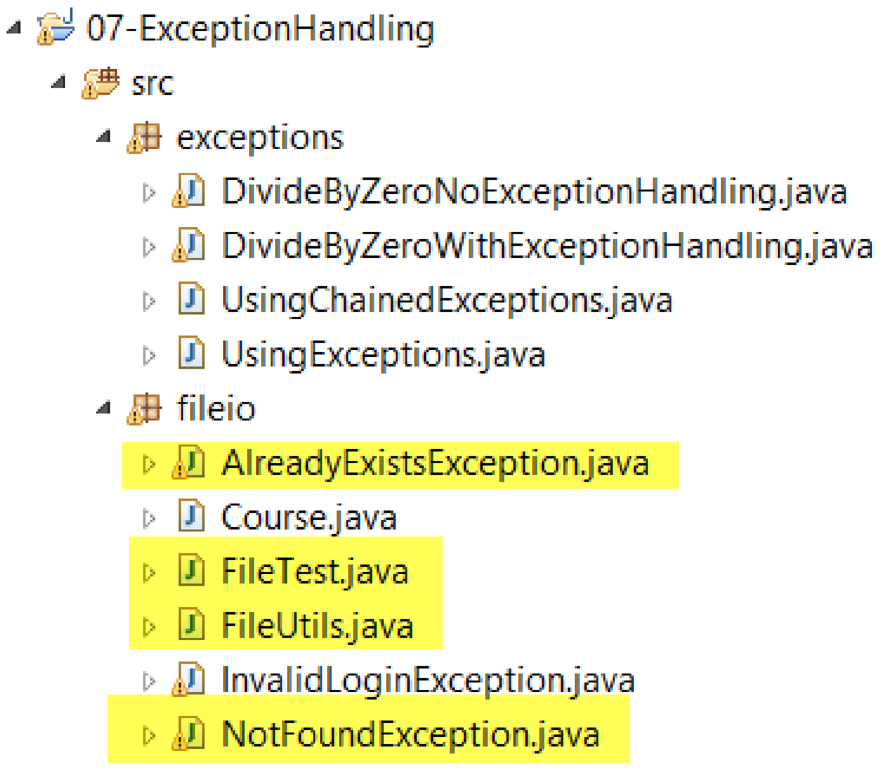
```
public void setRadius(double radius)
      throws IllegalArgumentException {
    if (radius >= 0)
      radius =  radius;
    else
      throw new IllegalArgumentException(
        "Radius cannot be negative");
}
```

Notes

- A method can throw multiple exceptions

```
public static void appendToFile(String filePath, String textToAppend)
        throws IOException, AlreadyExistsException {
```

- See the posted examples for further details



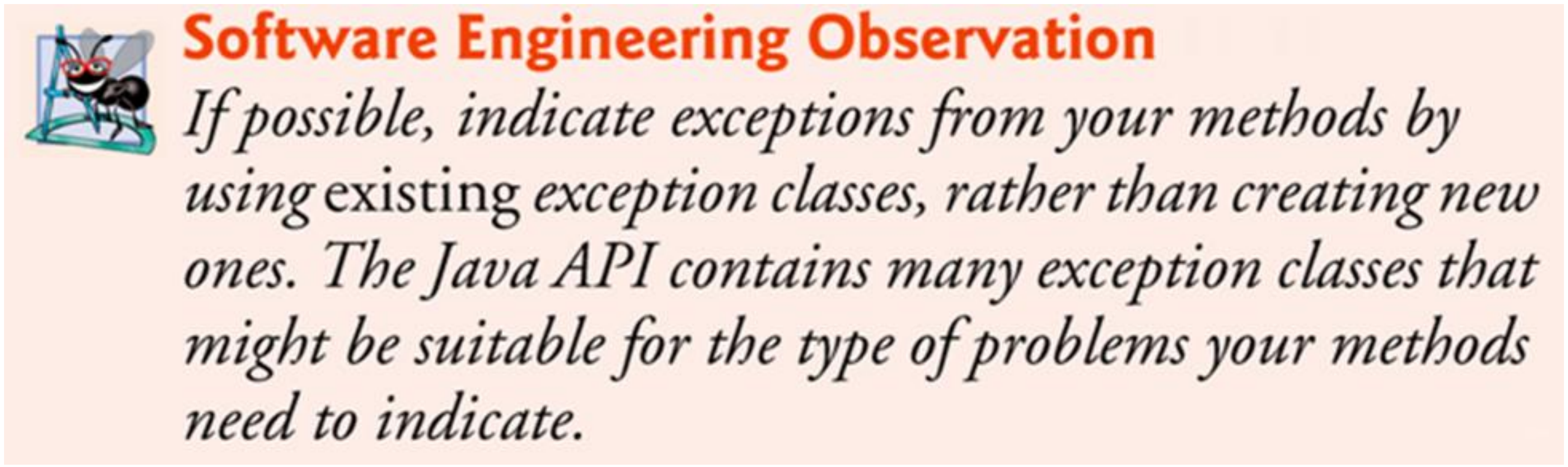
```
07-ExceptionHandling
  src
    exceptions
      DivideByZeroNoExceptionHandling.java
      DivideByZeroWithExceptionHandling.java
      UsingChainedExceptions.java
      UsingExceptions.java
    fileio
      AlreadyExistsException.java
      Course.java
      FileTest.java
      FileUtils.java
      InvalidLoginException.java
      NotFoundException.java
```

Custom Exceptions

Back

Declaring New Exception Types

- Sometimes it's useful to declare your own exception classes that are specific to the problems that can occur when another programmer uses your reusable classes.
- A new exception class must extend an existing exception class to ensure that the class can be used with the exception-handling mechanism.



Software Engineering Observation

If possible, indicate exceptions from your methods by using existing *exception classes, rather than creating new ones. The Java API contains many exception classes that might be suitable for the type of problems your methods need to indicate.*

Examples

```java
public class InvalidLoginException extends Exception {
    //An example of throwing it could be:
    //throw new InvalidLoginException("Email and/or password are invalid");
    public InvalidLoginException(String message) {
        super(message);
    }

    public InvalidLoginException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

```java
public class AlreadyExistsException extends Exception {
    private String dataToAdd; //with getters and setters
    private String destination;
    public AlreadyExistsException(String message) {
        super(message);
    }

    public AlreadyExistsException(String message, String dataToAdd, String destination) {
        super(message);
        this.dataToAdd = dataToAdd;
        this.destination = destination;
    }

    public AlreadyExistsException(String message, Throwable cause) {
        super(message, cause);
    }
```

A custom Exception is a class that extends Exception.
It can have extra attributes and methods.

When to Use Exceptions

- Use it if the event is truly exceptional and is an error
- Do not use it to deal with simple, expected situations.
- Example:

```
try {
   System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
   System.out.println("refVar is null");
}
```

Can be replaced by:

```
if (refVar != null)
   System.out.println(refVar.toString());
else
   System.out.println("refVar is null");
```

Summary

- Exceptions are a powerful mechanism for separating Error-Handling Code from "Regular" Code => **this simplifies the normal flow code.**
- The ***try block*** identifies a block of code in which an exception can occur.
- The ***catch block*** identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- The ***finally block*** identifies a block of code that is guaranteed to execute, and is the right place to release resources acquired in the try block such as closing files, database connections and network connections.