

AP Computer Science B

Java Object-Oriented Programming [Ver. 2.0]

Unit 4: Object-Oriented Design

WEEK 9: CHAPTER 14 EXCEPTIONS AND FILE I/O

DR. ERIC CHOU

IEEE SENIOR MEMBER



Objectives

- How Exceptions happen?
- Exception Types and Exception Handling
- Custom-Designed Exception Class
- Basic File Class and File I/O
- Input Validation
- File Validation

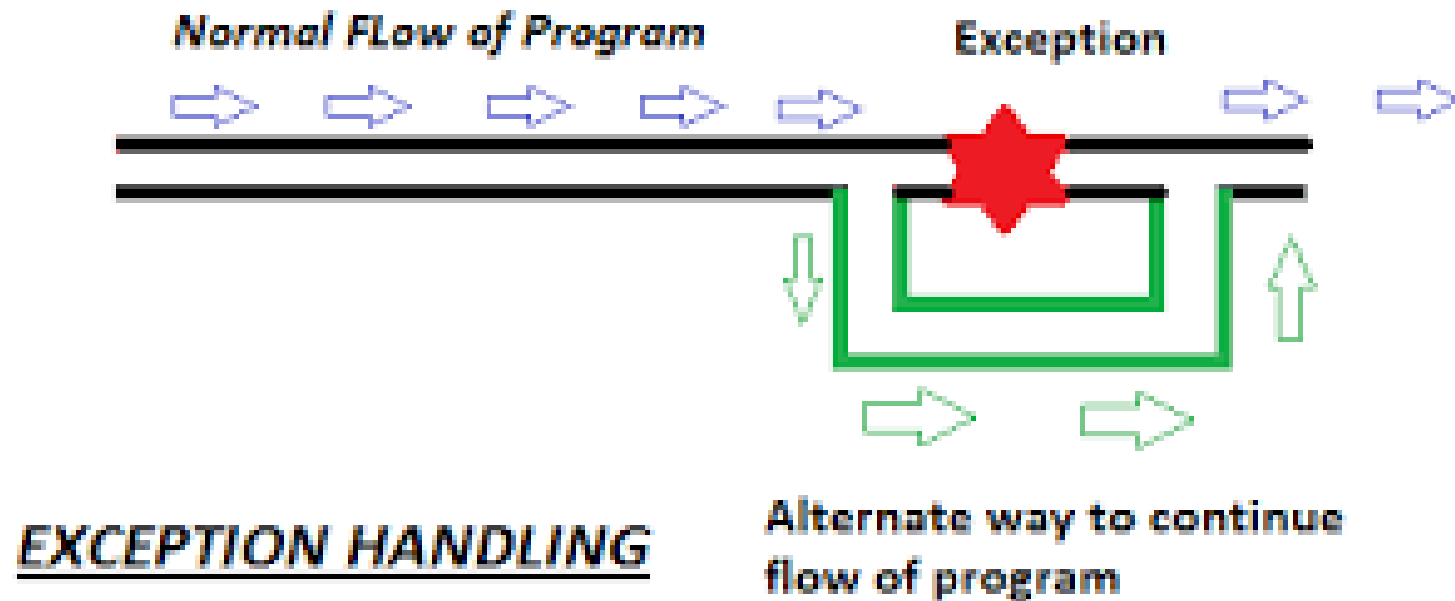


Overview

LECTURE 1



Exceptions





Introduction

Exception Handling enables a program to deal with exceptional situations and continue its normal execution.

- Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out.
- For example, if you access an array using an index that is out of bounds, you will get a runtime error with an **ArrayIndexOutOfBoundsException**.
- If you enter a double value when your program expects an integer, you will get a runtime error with an **InputMismatchException**.



Introduction

Exception Handling enables a program to deal with exceptional situations and continue its normal execution.

- In Java, runtime errors are thrown as exceptions. **An exception is an object that represents an error or a condition that prevents execution from proceeding normally.**
- Exception Classes are information classes like Class class returned by getClass().
- If the exception is not handled, the program will terminate abnormally. How can you handle the exception so that the program can continue to run or else terminate gracefully? This chapter introduces this subject and text input and output.



Issues Involves Exceptions

The **exception handling** in java is one of the powerful mechanism to handle the **runtime** errors so that normal flow of the application can be maintained.

- Why it happens and where does it happens?
- How to raise an exception and how to catch it?
- How to handle it?



Exception-Handling Overview

Exemplary Run-Time Errors

- Show runtime error (**Quotient.java**) occurs when division by 0.
- Fix it using an if statement (**QuotientWithIf.java**) (check and prevent the division by 0 to happen)
- Using method to quarantine the division error (**QuotientWithMethod.java**)
- Introduce try-catch (**QuotientWithException.java**)



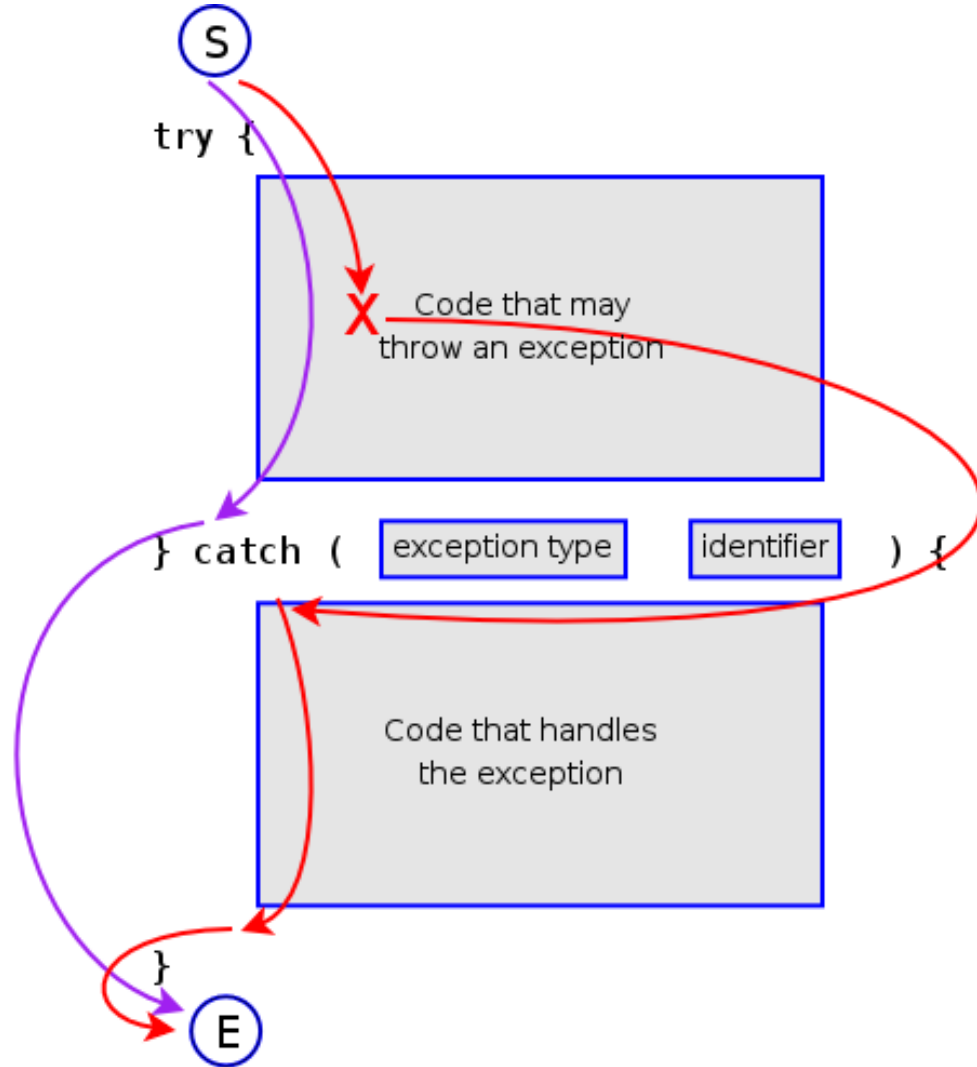
Exception Advantages

Handle run-time errors with Exceptions.

(QuotientWithException.java)

- Now you see the *advantages* of using exception handling. It enables a method to throw an exception to its caller.
- Without this capability, a method must handle the exception or terminate the program.

Exception throws and catch



- Exception classes are **information** class which contains types of exception and the other information.



From the Method to Exception Handling

The method `quotient` returns the quotient of two integers. If `number2` is `0`, it cannot return a value, so the program is terminated. This is clearly a problem. You should not let the method terminate the program—the *caller* should decide whether to terminate the program.

How can a method notify its caller an exception has occurred? Java enables a method to throw an exception that can be caught and handled by the caller. Listing 12.3 can be rewritten, as shown in `QuotientWithException.java`.

If `number2` is `0`, the method throws an exception by executing

```
throw new ArithmeticException("Divisor cannot be zero");
```

The value thrown, in this case `new ArithmeticException("Divisor cannot be zero")`, is called an *exception*. The execution of a `throw` statement is called *throwing an exception*. The exception is an object created from an exception class. In this case, the exception class is `java.lang.ArithmeticException`. The constructor `ArithmeticException(str)` is invoked to construct an exception object, where `str` is a message that describes the exception.



Exception Handling

- When an exception is thrown, the normal execution flow is interrupted. As the name suggests, to “throw an exception” is to pass the exception from one place to another. The statement for invoking the method is contained in a **try** block and a **catch** block. The **try** block contains the code that is executed in normal circumstances. The exception is caught by the **catch** block. The code in the **catch** block is executed to *handle the exception*.
- Afterward, the statement after the **catch** block is executed. The **throw** statement is analogous to a method call, but instead of calling a method, it calls a **catch** block. In this sense, a **catch** block is like a method definition with a parameter that matches the type of the value being thrown. Unlike a method, however, after the **catch** block is executed, the program control does not return to the **throw** statement; instead, it executes the next statement after the **catch** block.



Exception Handling

like calling a method

The identifier **ex** in the **catch**–block header

```
catch (ArithmeticException ex)
```

acts very much like a parameter in a method. Thus, this parameter is referred to as a **catch**–block parameter. The type (e.g., **ArithmeticException**) preceding **ex** specifies what kind of exception the **catch** block can catch. Once the exception is caught, you can access the thrown value from this parameter in the body of a **catch** block.



Exception Handling

like calling a method

- In summary, a template for a **try-throw-catch** block may look like this:

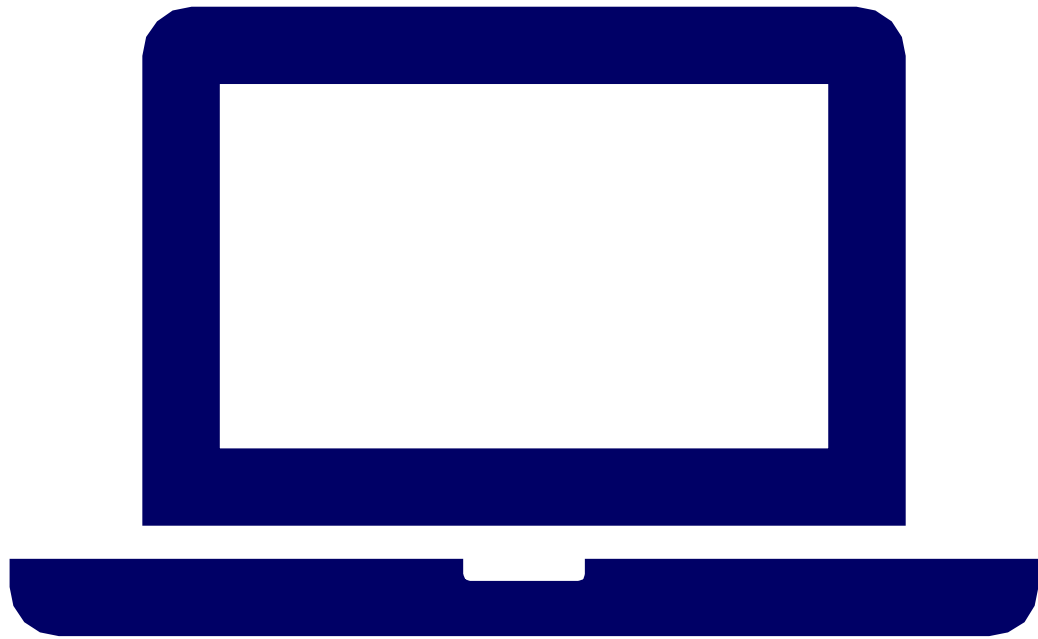
```
try {  
    Code to run;  
    A statement or a method that may throw an exception;  
    More code to run;  
}  
catch (type ex) {  
    Code to process the exception;  
}
```

- An exception may be thrown directly by using a **throw** statement in a **try** block, or by invoking
- a method that may throw an exception.



Handling InputMismatchException

- Errors when input data type mismatched
(**InputMismatchException.java**)
- By handling InputMismatchException, your program will continuously read an input until it is correct.



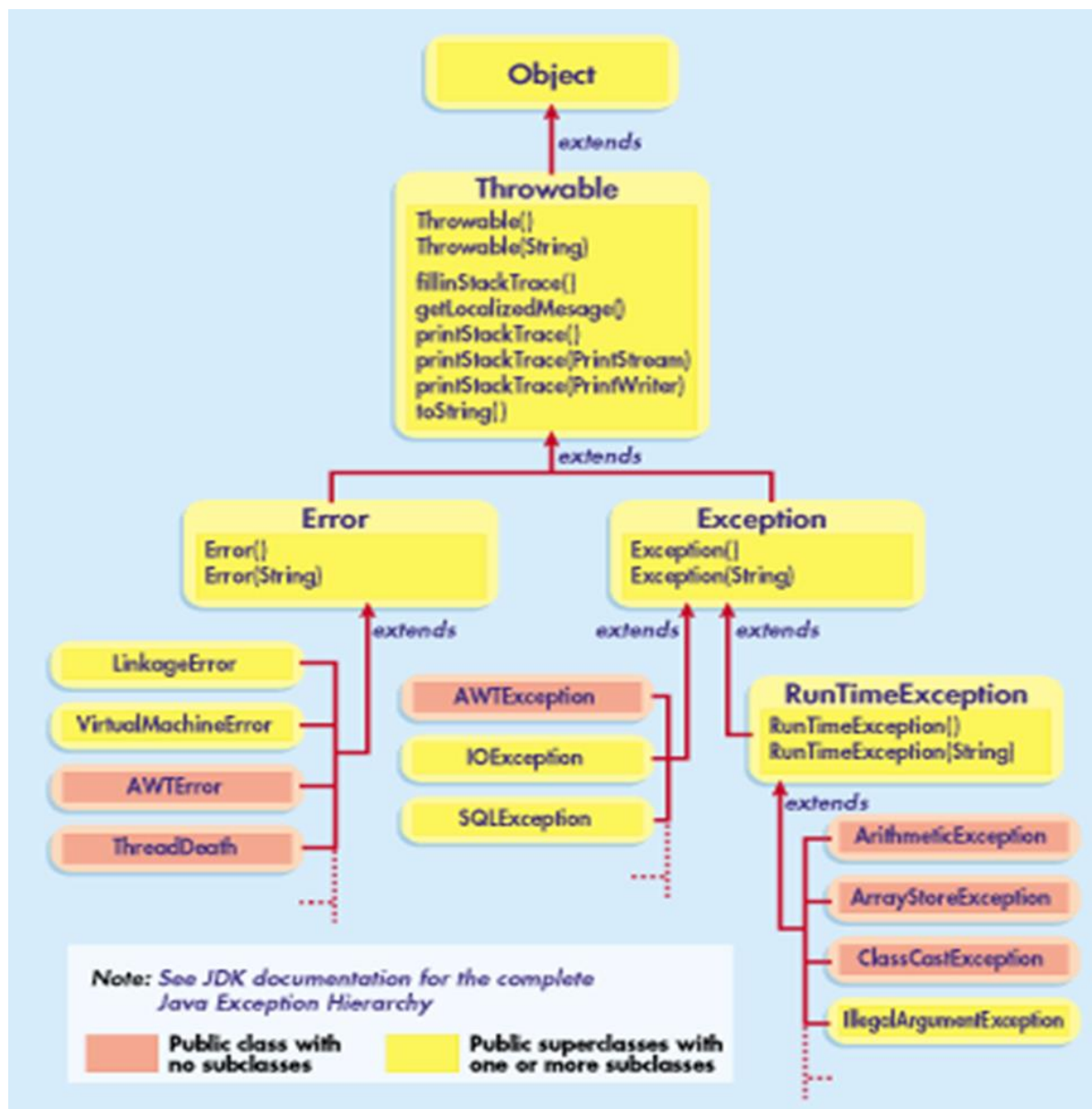
Demonstration Program

INPUTMISMATCHEXCEPTION.JAVA

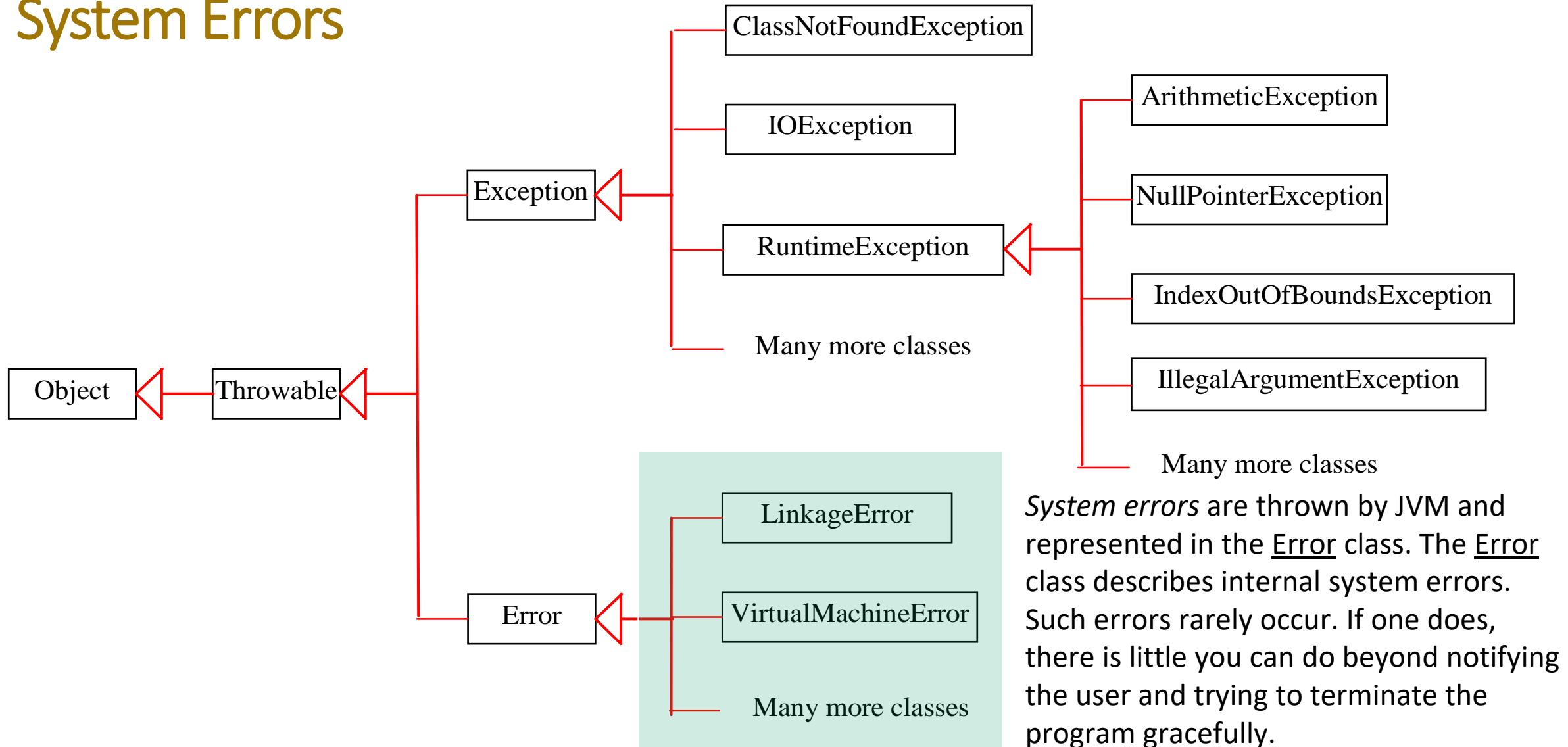


Exception Types

LECTURE 2



System Errors



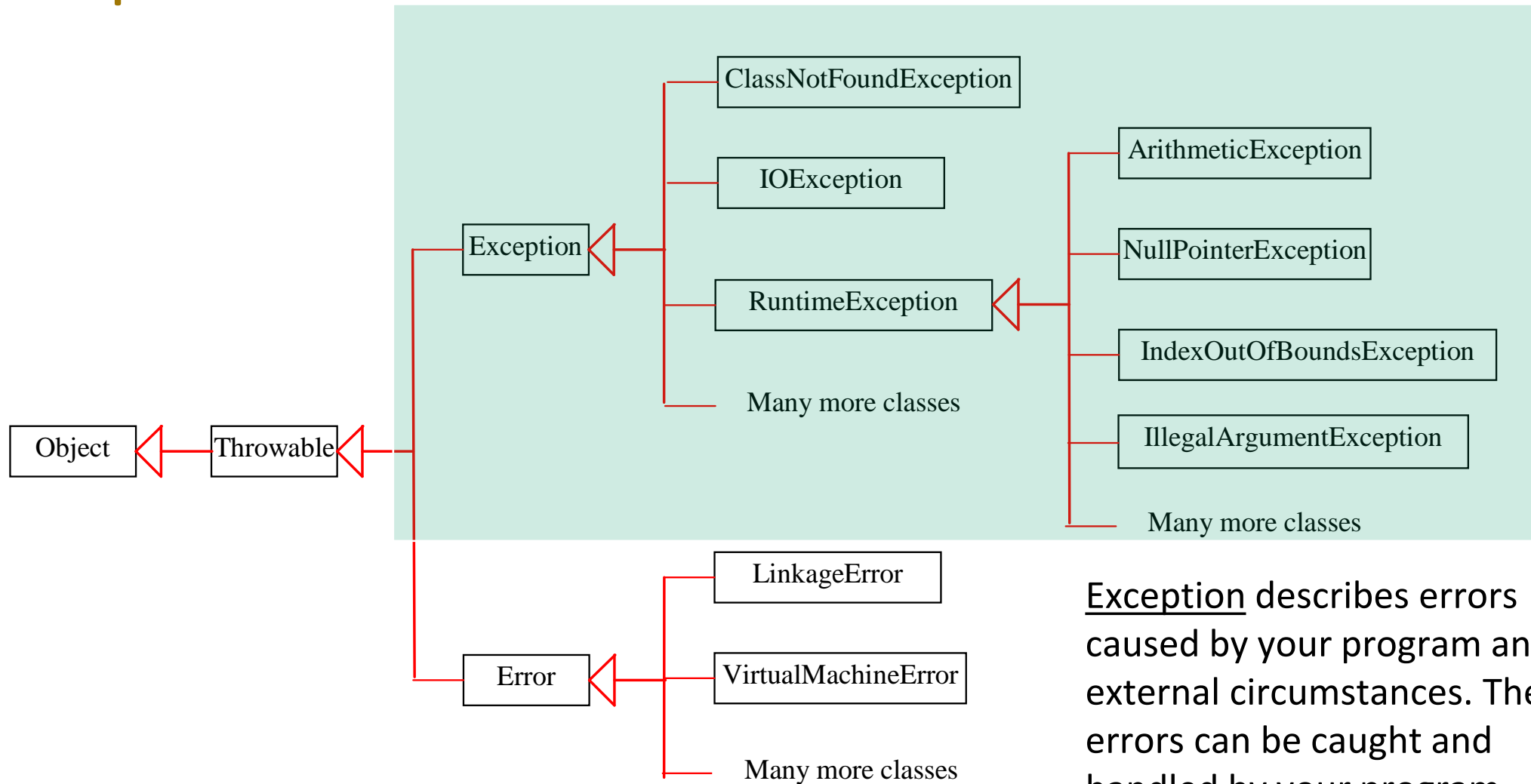


System Error Classes

TABLE 12.1 Examples of Subclasses of **Error**

<i>Class</i>	<i>Reasons for Exception</i>
LinkageError	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
VirtualMachineError	The JVM is broken or has run out of the resources it needs in order to continue operating.

Exceptions



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

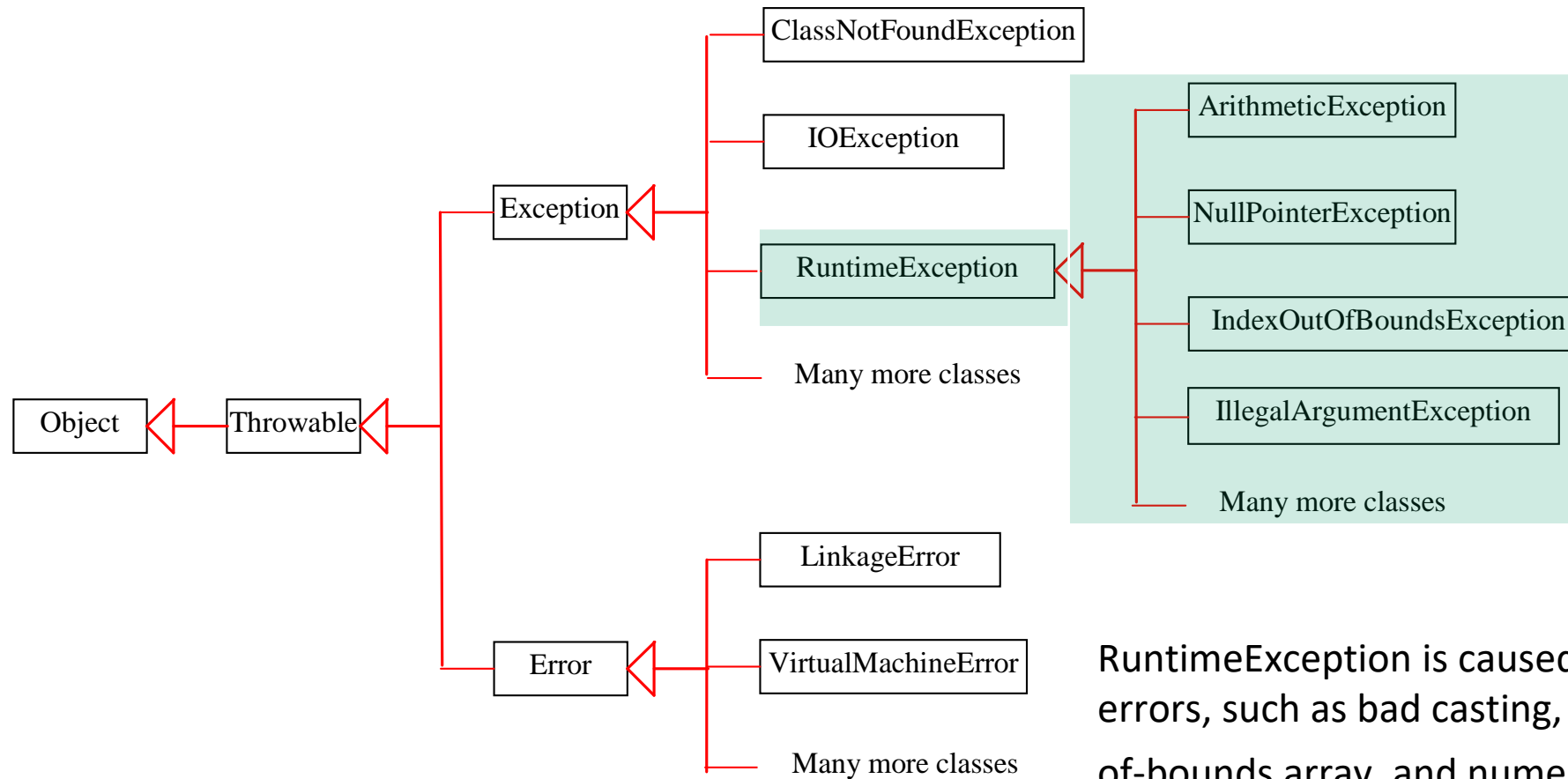


Exception

TABLE 12.2 Examples of Subclasses of **Exception**

<i>Class</i>	<i>Reasons for Exception</i>
ClassNotFoundException	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the <code>java</code> command, or if your program were composed of, say, three class files, only two of which could be found.
IOException	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of IOException are InterruptedIOException , EOFException (EOF is short for End of File), and FileNotFoundException .

Runtime Exceptions



`RuntimeException` is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.



RuntimeException

TABLE 12.3 Examples of Subclasses of `RuntimeException`

<i>Class</i>	<i>Reasons for Exception</i>
<code>ArithmeticException</code>	Dividing an integer by zero. Note that floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
<code>NullPointerException</code>	Attempt to access an object through a <code>null</code> reference variable.
<code>IndexOutOfBoundsException</code>	Index to an array is out of range.
<code>IllegalArgumentException</code>	A method is passed an argument that is illegal or inappropriate.



Checked Exceptions vs. Unchecked Exceptions

- **RuntimeException, Error** and their subclasses are known as *unchecked exceptions*. (Compiler won't force you to check it.)
- All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions. (Such as IOException, if you do nothing about it, compiler won't let you pass.)



Unchecked Exceptions

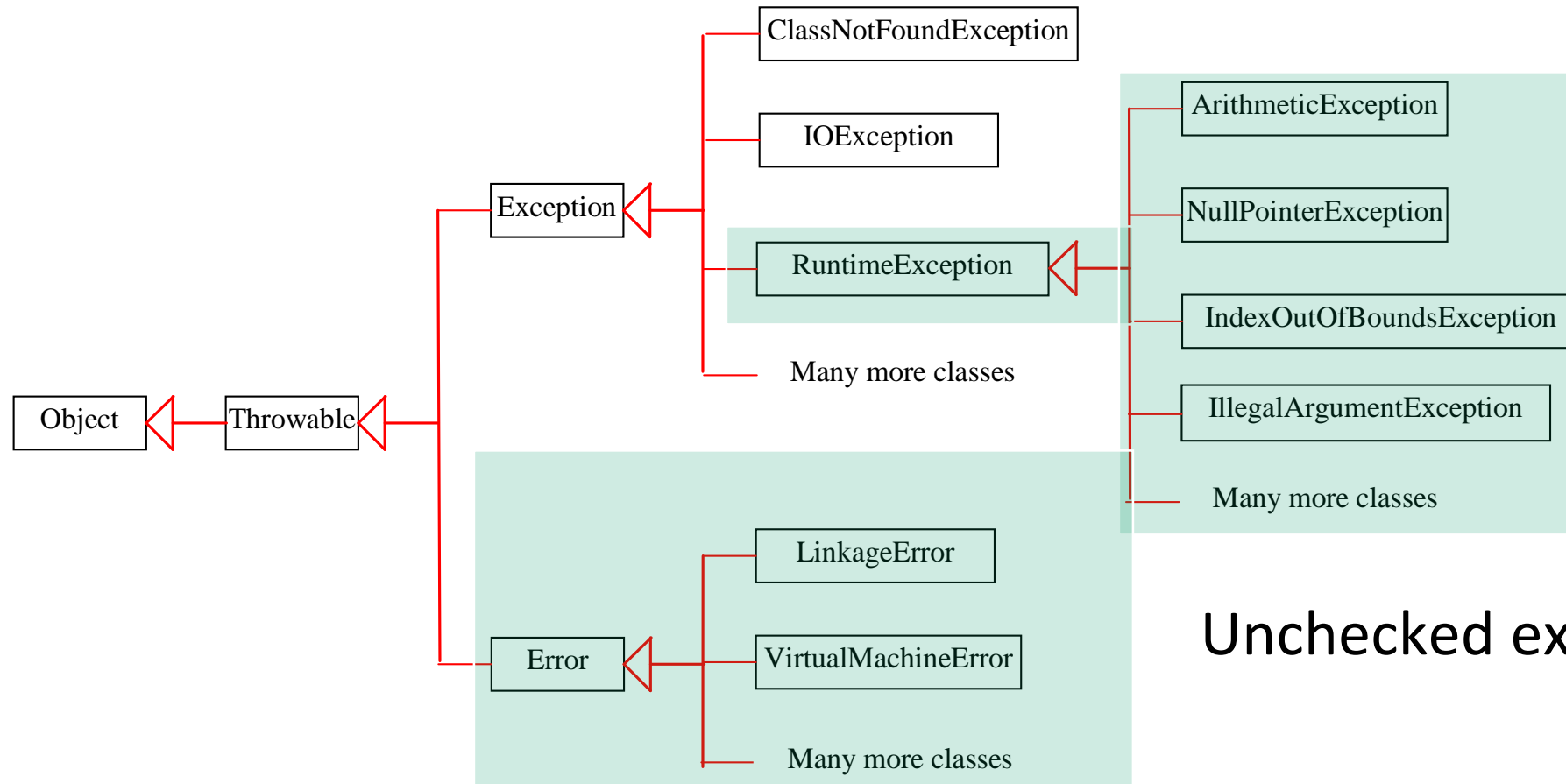
- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable.
- For example, a **NullPointerException** is thrown if you access an object through a reference variable before an object is assigned to it; an **IndexOutOfBoundsException** is thrown if you access an element in an array outside the bounds of the array.



Unchecked Exceptions

- These are the logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in the program.
- To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

Unchecked Exceptions



Unchecked exception.



Exception Handling

LECTURE 3



Exception & Call Stack

- When an exception occurs inside a Java method, the method creates an **Exception** object and passes the **Exception** object to the **JVM** (in Java term, the method "**throw**" an Exception).
- The Exception object contains the type of the exception, and the state of the program when the exception occurs.



Exception & Call Stack

- The JVM is responsible for finding an exception handler to process the Exception object. It searches backward through the call stack until it finds a matching exception handler for that particular class of Exception object (in Java term, it is called "catch" the Exception).
- **If the JVM cannot find a matching exception handler in all the methods in the call stack, it terminates the program.**

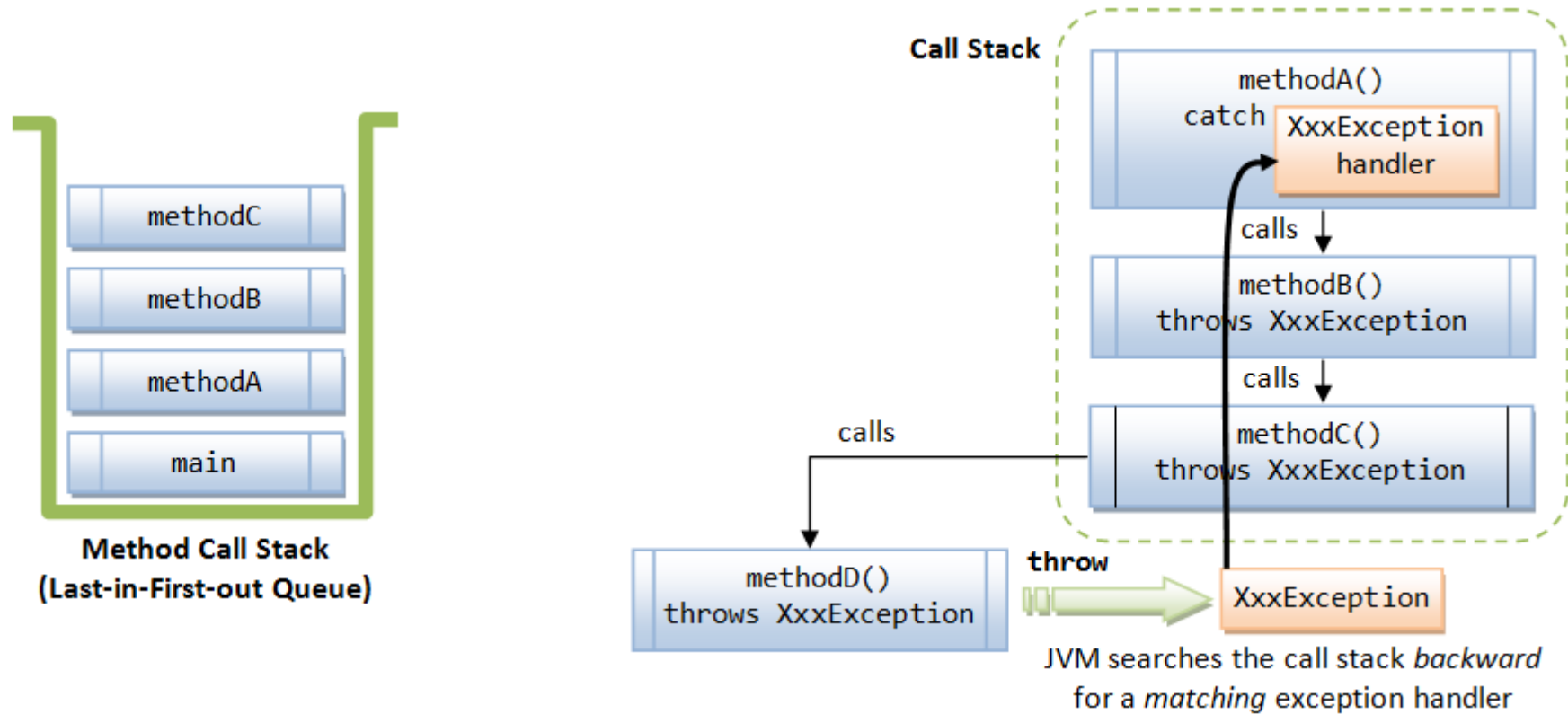


Exception & Call Stack

- This process is illustrated as follows. Suppose that **methodD()** encounters an abnormal condition and throws a **XxxException** to the JVM. The **JVM** searches **backward** through the call stack for a **matching exception handler**. It finds **methodA()** having a **XxxException** handler and passes the exception object to the handler.
- Notice that **methodC()** and **methodB()** are required to declare "throws **XxxException**" in their method signatures in order to compile the program. (Relay of throwing exceptions)

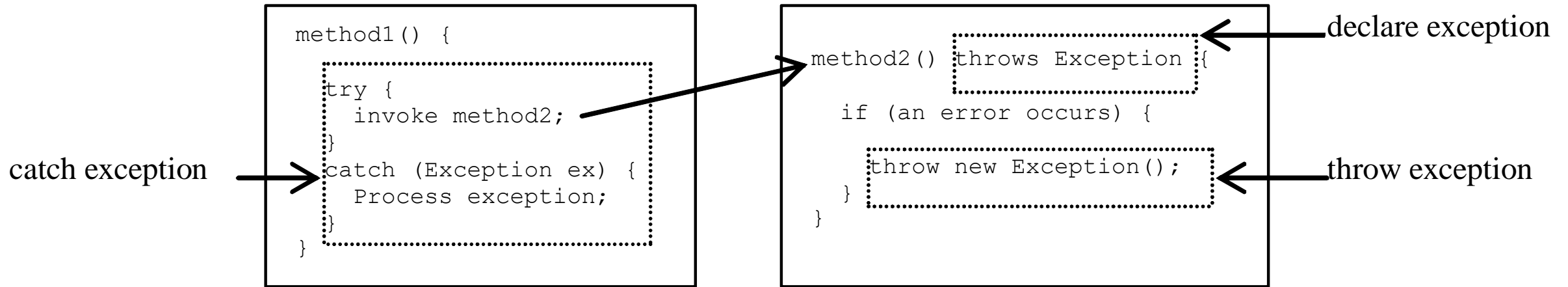


Exception Handling Mechanism





Declaring, Throwing, and Catching Exceptions



(The exception can be checked or un-checked.)



Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()  
    throws IOException
```

```
public void myMethod()  
    throws IOException, OtherException
```



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,

```
throw new TheException();
```

```
TheException ex = new TheException();  
throw ex;
```



Throwing Exceptions Example

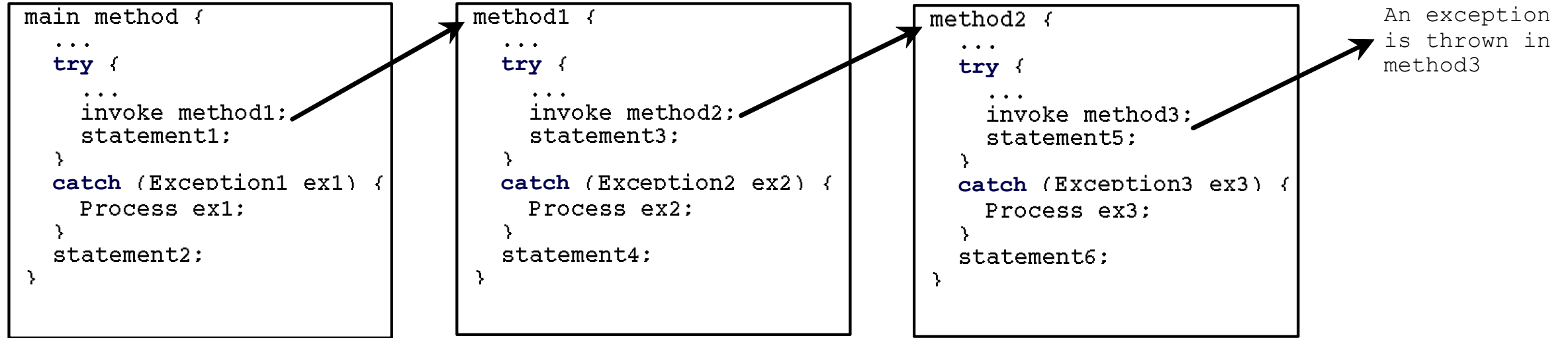
```
/** Set a new radius */  
public void setRadius(double newRadius)  
    throws IllegalArgumentException {  
    if (newRadius >= 0)  
        radius = newRadius;  
    else  
        throw new IllegalArgumentException(  
            "Radius cannot be negative");  
}
```



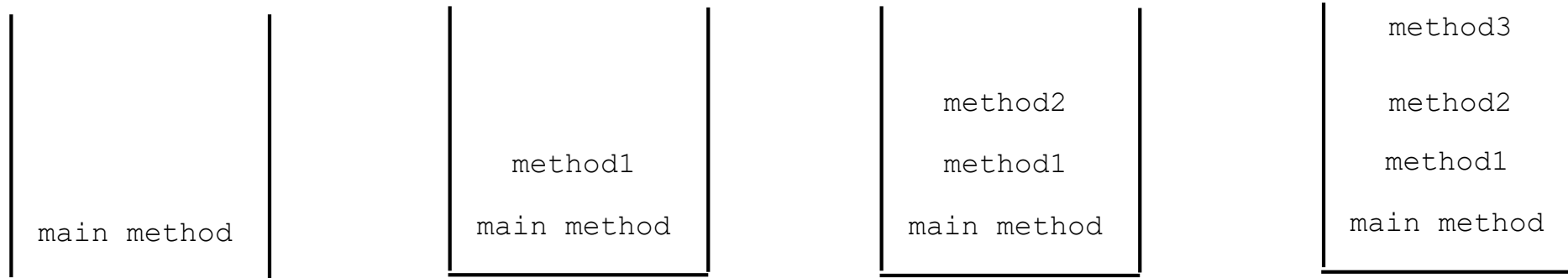
Catching Exceptions

```
try {  
    statements;    // Statements that may throw exceptions  
}  
catch (Exception1 exVar1) {  
    handler for exception1;  
}  
catch (Exception2 exVar2) {  
    handler for exception2;  
}  
...  
catch (ExceptionN exVar3) {  
    handler for exceptionN;  
}
```

Catching Exceptions



Call Stack



Throwable Class and Exception Class

java.lang.Throwable

+getMessage(): String
+toString(): String
+printStackTrace(): void
+getStackTrace():
Stack Trace Element[]

Returns the message that describes this exception object.

Returns the concatenation of three strings: (1) the full name of the exception class; (2) ":" (a colon and a space); (3) the `getMessage()` method.

Prints the `Throwable` object and its call stack trace information on the console.

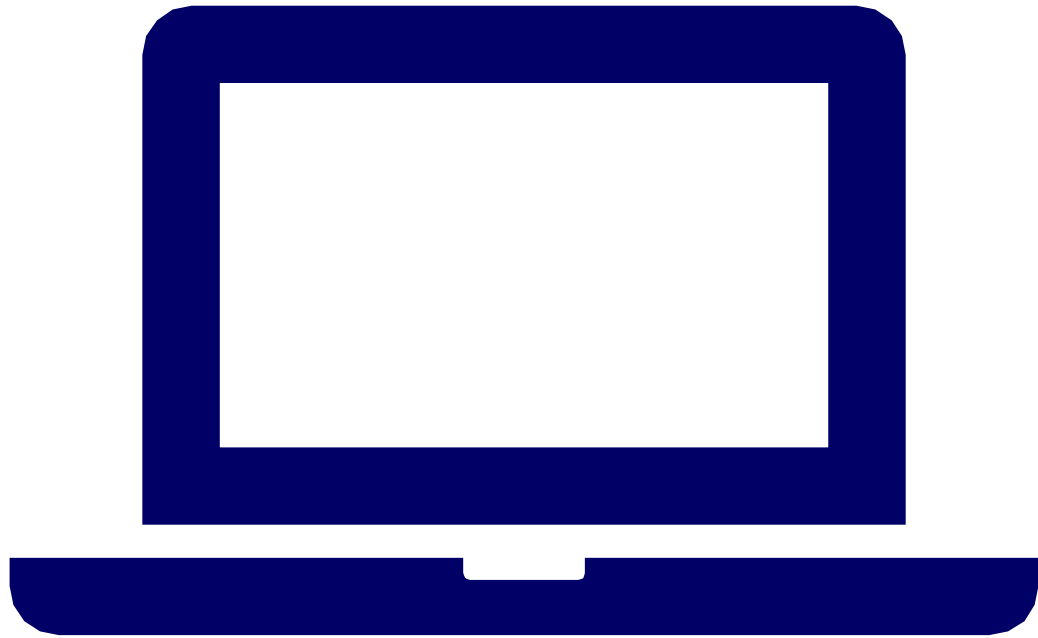
Returns an array of stack trace elements representing the stack trace pertaining to this exception object.

java.lang.Exception

+Exception()
+Exception(message: String)

Constructs an exception with no message.

Constructs an exception with the specified message.



Demonstration Program

TESTEXCEPTION.JAVA



Catch or Declare Checked Exceptions

- Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose that method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException), you have to write the code as shown in (a) or (b).

```
void p1() {  
    try {  
        p2();  
    }  
    catch (IOException ex) {  
        ...  
    }  
}
```

(a)

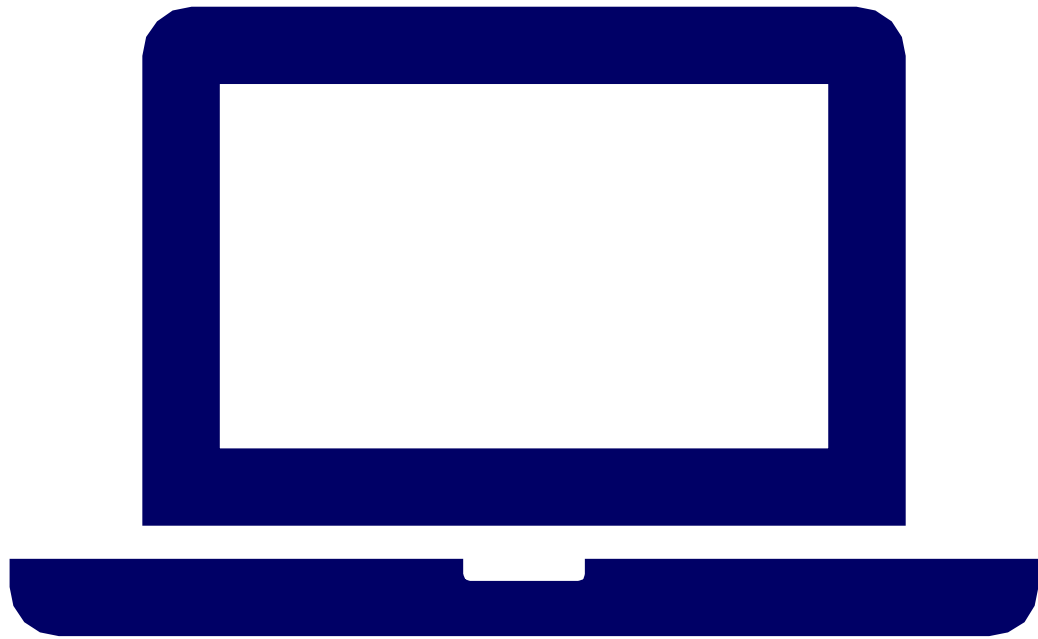
```
void p1() throws IOException {  
    p2();  
}
```

(b)



Example: Declaring, Throwing, and Catching Exceptions

- **Objective:** This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in Chapter 8. The new setRadius method throws an exception if radius is negative.



Demonstration Program

CIRCLEWITHEXCEPTION.JAVA
TESTCIRCLEWITHEXCEPTION.JAVA



Advanced Topics for Exception Handling

LECTURE 4

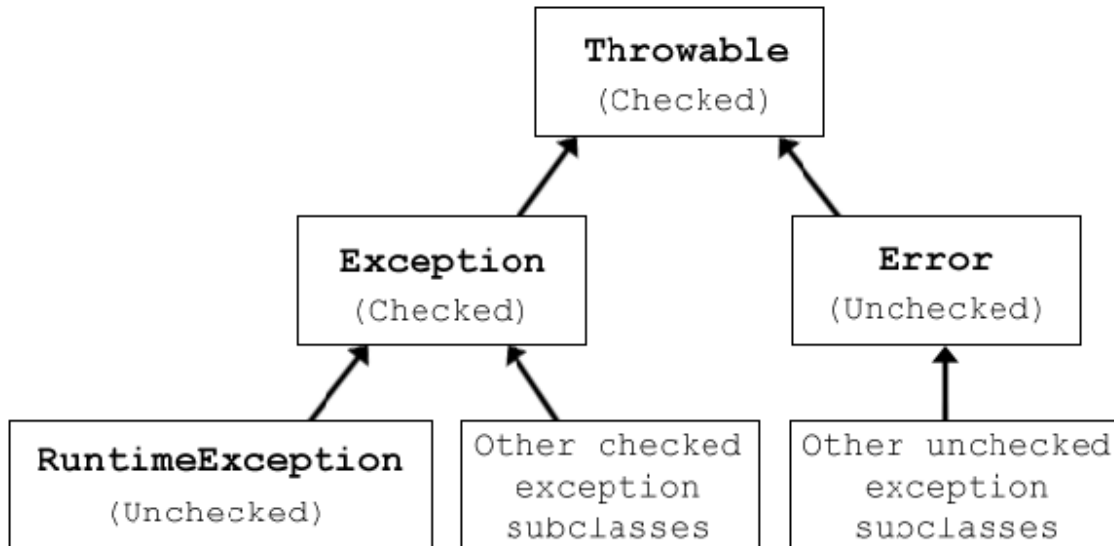
Summary for Exception so Far

Unchecked Exceptions:

1. Use if-statement to avoid.
2. When happened, programs stop.

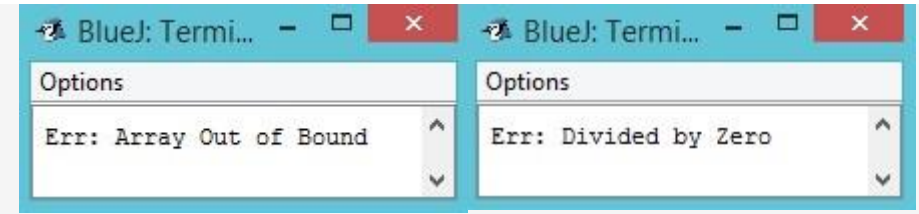
Checked Exceptions:

1. try-catch-block with some handler.
2. method() throws Exception to escalate the exceptions to upper level methods
3. If no proper handler, program stop, otherwise, continue after the handler.



Java Multiple Catch Blocks

```
public class MultipleExceptionExample {  
    public static void main(String argv[]) {  
        int num1 = 10; int num2 = 0; int result = 0;  
        int arr[] = new int[5];  
        try {  
            arr[0] = 0; arr[1] = 1; arr[2] = 2; arr[3] = 3; arr[4] = 4; arr[5] = 5;  
            result = num1 / num2;  
            System.out.println("Result of Division : " + result);  
        }  
        catch(ArithmeticException e) { System.out.println("Err: Divided by Zero");}  
        catch(ArrayIndexOutOfBoundsException e)  
            { System.out.println("Err: Array Out of Bound"); }  
    }  
}
```



```
//comment out the array  
// element assignments
```



The finally Clause

The finally clause is always executed regardless whether an exception occurred or not.

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

The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting **cleanup code** in a finally block is always a good practice, even when no exceptions are anticipated.

Trace a Program Execution

Suppose no
exceptions in the
statements

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

Next statement;

Trace a Program Execution

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

The final block is
always executed

Next statement;

Trace a Program Execution

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

Next statement;

Next statement in the
method is executed

A green callout box with a pointer indicating the next statement to be executed. The box is green with a white border and contains the text "Next statement in the method is executed". A green arrow points from the box to the "Next statement;" text.

Trace a Program Execution

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

Suppose an exception
of type Exception1 is
thrown in statement2

Next statement;

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.

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
catch (Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```

statement2 throws an exception of type Exception2.

Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
catch (Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```

Handling exception



Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
catch (Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```

Execute the final block



Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
catch (Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;



Rethrow the exception
and control is
transferred to the caller



Cautions When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify. Be aware, however, that **exception handling usually requires more time and resources** because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.



When to Throw Exceptions

- An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you can handle the exception in the method where it occurs, there is no need to throw it.



When to Use Exceptions

When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code

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



When to Use Exceptions

is better to be replaced by

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



Rethrowing Exceptions

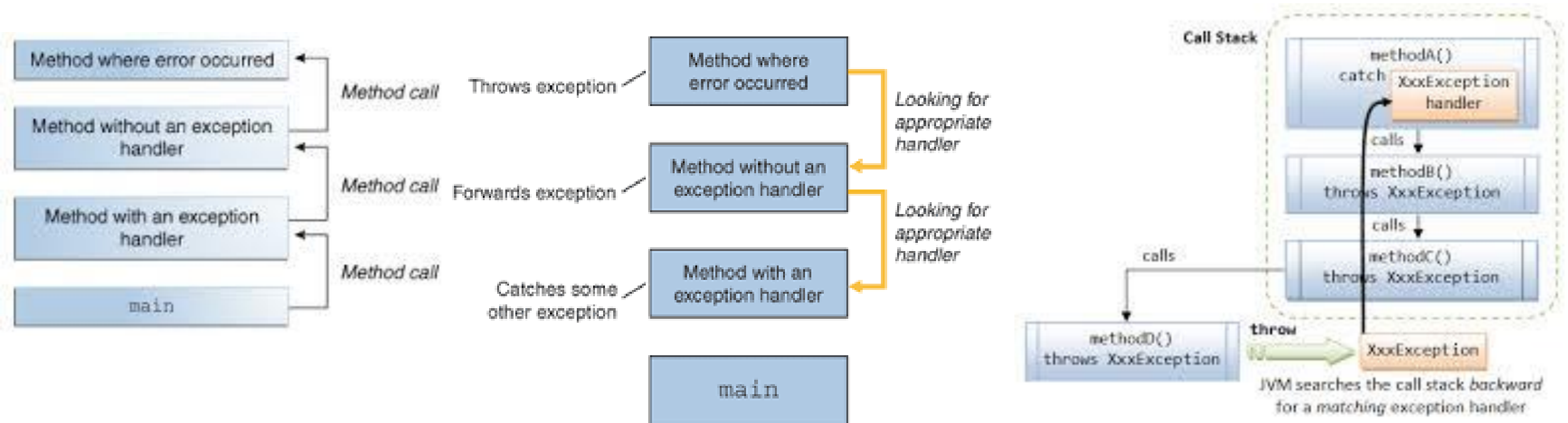
Java allows an exception handler to rethrow the exception if the handler cannot process the exception or simply wants to let **its caller** be notified of the exception.

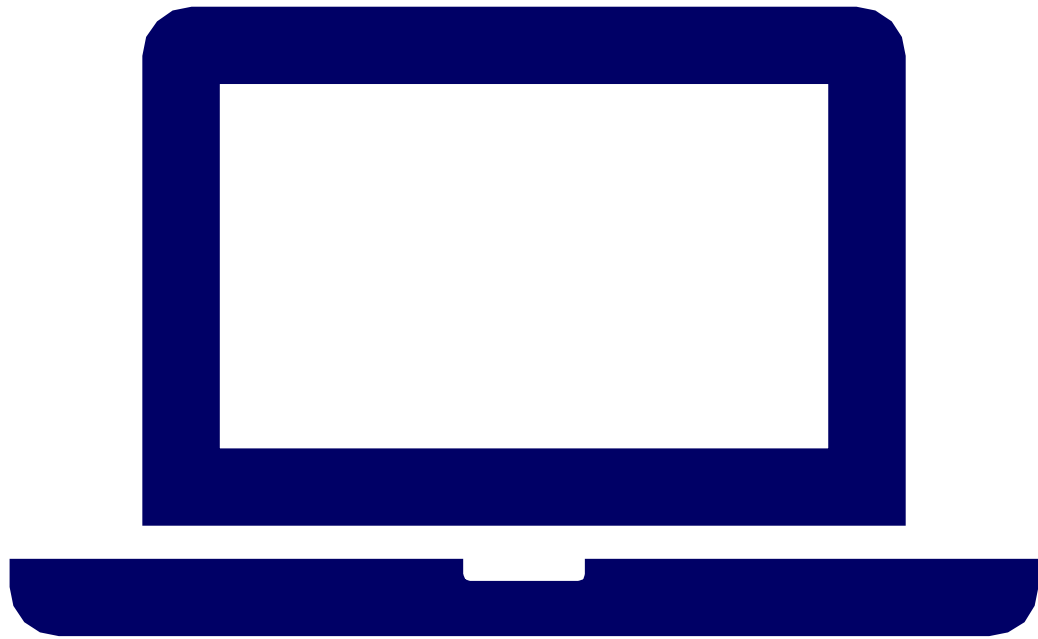
```
try {    // move up the higher method
        // level for other handler
    statements;
}
catch (TheException ex) {
    perform operations before exits;
    throw ex;
}
```




Chained Exceptions

Throwing an exception along with another exception forms a chained exception.





Demonstration Program

CHAINEDEXCEPTIONDEMO.JAVA



BlueJ: Terminal Window - Chapter12

Options

```
java.lang.Exception: New info from method1
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:24)
    at ChainedExceptionDemo.main(ChainedExceptionDemo.java:12)
    at __SHELL9.run(__SHELL9.java:6)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:497)
    at bluej.runtime.ExecServer$3.run(ExecServer.java:730)
Caused by: java.lang.Exception: New info from method2
    at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:29)
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:21)
    ... 7 more
```

```
public static void main(String[] args) {
    try {
        method1();
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
}

public static void method1() throws Exception {
    try {
        method2();
    }
    catch (Exception ex) {
        throw new Exception("New info from method1", ex);
    }
}

public static void method2() throws Exception {
    throw new Exception("New info from method2");
}
```

main with handler
ex.printStackTrace

method1 no handler
rethrows

method2 throws ex



Defining Custom Exception Classes

LECTURE 5



Defining Custom Exception Classes

You can define a custom exception class by extending the `java.lang.Exception` class

<code>java.lang.Exception</code>	
<code>+Exception()</code> <code>+Exception(message: String)</code>	Constructs an exception with no message. Constructs an exception with the specified message.

- Java provides quite a few exception classes. Use them whenever possible instead of defining your own exception classes. However, if you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from **Exception** or from a subclass of **Exception**, such as **IOException**.



Defining Custom Exception Classes

- Use the exception classes in the API whenever possible.
- Define custom exception classes if the predefined classes are not sufficient.
- Define custom exception classes by extending Exception or a subclass of Exception.



Custom Exception Class Example

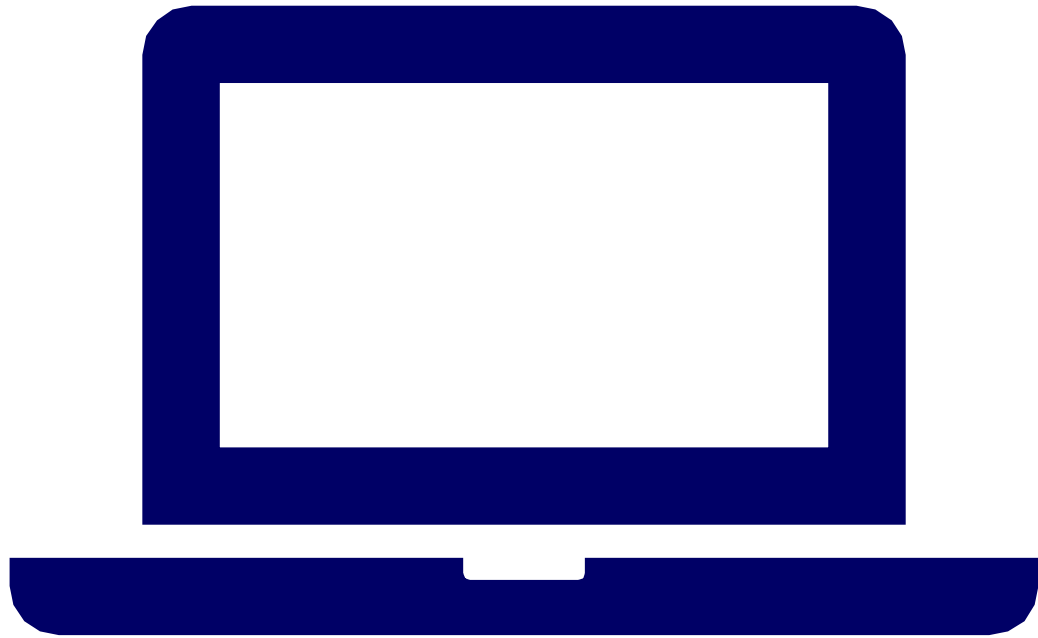
In `TestCircleWithException.java`, the `setRadius` method throws an exception if the radius is **negative**. Suppose you wish to pass the radius to the handler, you have to create a custom exception class.

Demo Programs:

`InvalidRadiusException.java`

`TestCircleWithCustomException.java`

(`CircleWithCustomException` and `TestCircleWithCustomException` classes)



Demonstration Program

INVALIDRADIUSException.JAVA
TESTCIRCLEWITHCUSTOMException.JAVA



Demo Program

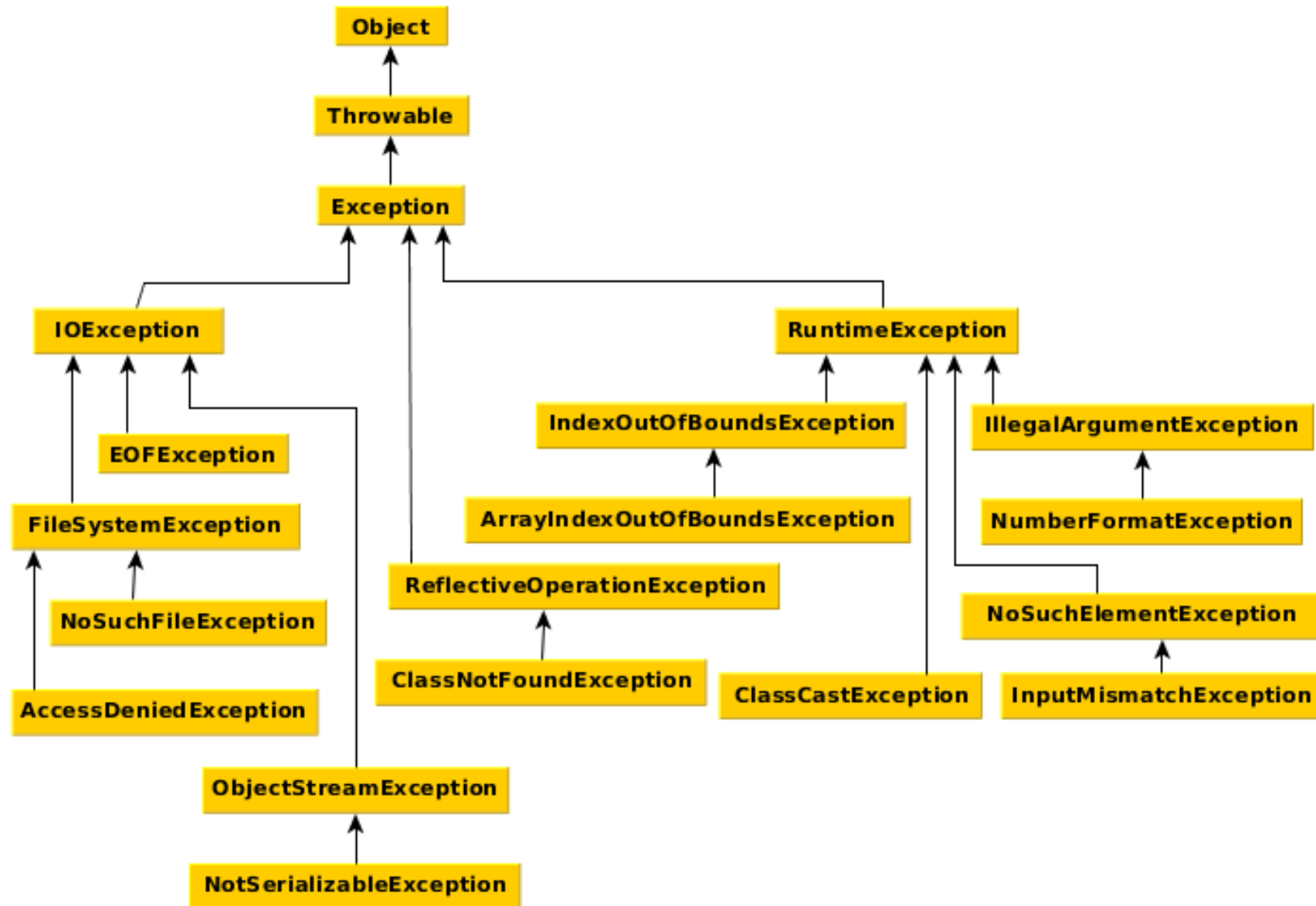
exceptions package

(exceptions.UserDefined.java/exceptions.ThrowsClause.java)

- The following demo program illustrates the creation of user-defined Exception classes. Look in the package

exceptions

- In addition to illustrating user-defined exception types, we're making a pedantic point of our knowledge that the **Double.parseDouble** function generates an exception type which is a subclass of **RuntimeException**; in general you should make the exception a closer match to the exception generated by the scanner.





Assertions (Non-AP)

LECTURE 6



Assertions

- An assertion is a Java statement that enables you to assert an assumption about your program. An assertion contains a Boolean expression that should be true during program execution. Assertions can be used to assure program correctness and avoid logic errors.



Declaring Assertions

An *assertion* is declared using the new Java keyword assert in JDK 1.4 as follows:

assert assertion; or
assert assertion : detailMessage;

where **assertion** is a Boolean expression and *detailMessage* is a primitive-type or an Object value.



Executing Assertions

- When an assertion statement is executed, Java evaluates the assertion. If it is false, an `AssertionError` will be thrown. The `AssertionError` class has a no-arg constructor and seven overloaded single-argument constructors of type `int`, `long`, `float`, `double`, `boolean`, `char`, and `Object`.
- For the first assert statement with no detail message, the no-arg constructor of `AssertionError` is used. For the second assert statement with a detail message, an appropriate `AssertionError` constructor is used to match the data type of the message. Since **`AssertionError`** is a subclass of **`Error`**, when an assertion becomes false, the program displays a message on the console and exits.



Executing Assertions Example

```
public class AssertionDemo {  
    public static void main(String[] args) {  
        int i; int sum = 0;  
        for (i = 0; i < 10; i++) {  
            sum += i;  
        }  
        assert i == 10;  
        assert sum > 10 && sum < 5 * 10 : "sum is " + sum;  
    }  
}
```



Compiling Programs with Assertions

Since assert is a new Java keyword introduced in JDK 1.4, you have to compile the program using a JDK 1.4 compiler. Furthermore, you need to include the switch **–source 1.4** in the compiler command as follows:

javac –source 1.4 AssertionDemo.java

NOTE: If you use JDK 1.5, there is no need to use the **–source 1.4** option in the command.



Running Programs with Assertions

By default, the assertions are disabled at runtime. To enable it, use the switch `-enableassertions`, or `-ea` for short, as follows:

```
java -ea AssertionDemo
```

Assertions can be selectively enabled or disabled at class level or package level. The disable switch is `-disableassertions` or `-da` for short. For example, the following command enables assertions in package package1 and disables assertions in class Class1.

```
java -ea:package1 -da:Class1 AssertionDemo
```



Using Exception Handling or Assertions

- Assertion should not be used to replace exception handling. Exception handling deals with unusual circumstances during program execution.
- Assertions are to assure the correctness of the program. Exception handling addresses robustness and assertion addresses correctness. Like exception handling, assertions are not used for normal tests, but for internal consistency and validity checks.
- Assertions are checked at runtime and can be turned on or off at startup time.



Using Exception Handling or Assertions, cont.

- *Do not use assertions for argument checking in public methods.* Valid arguments that may be passed to a public method are considered to be part of the method's contract. The contract must always be obeyed whether assertions are enabled or disabled. For example, the following code should be rewritten using exception handling.

```
public void setRadius(double newRadius) {  
    assert newRadius >= 0;  
    radius = newRadius;  
}
```



Using Exception Handling or Assertions, cont.

Use assertions to reaffirm assumptions. This gives you more confidence to assure correctness of the program. A common use of assertions is to replace assumptions with assertions in the code. (Check expected result or true for validity/invalidity)



Using Exception Handling or Assertions, cont.

Another good use of assertions is place assertions in a switch statement without a default case. For example,

```
switch (month) {  
    case 1: ... ; break;  
    case 2: ... ; break;  
    ...  
    case 12: ... ; break;  
    default: assert false : "Invalid month: " + month  
} // use assertion to notify programmer to take care of it
```

Assertion is replaced by JUnit Testing, Please try my course [Java JUnit for Unit Testing](#)



File Class

LECTURE 7



The File Class

The File class is intended to provide an abstraction that deals with most of the **machine-dependent** complexities of files and path names in a machine-independent fashion. The **filename** is a **string**. The File class is a wrapper class for the file name and its directory path.

java.io.File	
+File (pathname: String)	Creates a File object for the specified pathname. The pathname may be a directory or a file.
+File (parent: String, child: String)	Creates a File object for the child under the directory parent. The child may be a filename or a subdirectory.
+File (parent: File, child: String)	Creates a File object for the child under the directory parent. The parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of <i>hidden</i> is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On UNIX systems, a file is hidden if its name begins with a period (.) character.
+getAbsolutePath(): String	Returns the complete absolute file or directory name represented by the File object.
+getCanonicalPath(): String	Returns the same as getAbsolutePath() except that it removes redundant names, such as "." and "..", from the pathname, resolves symbolic links (on UNIX), and converts drive letters to standard uppercase (on Windows).
+getName(): String	Returns the last name of the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getName() returns test.dat.
+getPath(): String	Returns the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getPath() returns c:\\book\\test.dat.
+getParent(): String	Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File("c:\\book\\test.dat").getParent() returns c:\\book.
+lastModified(): long	Returns the time that the file was last modified.
+length(): long	Returns the size of the file, or 0 if it does not exist or if it is a directory.
+listFiles(): File[]	Returns the files under the directory for a directory File object.
+delete(): boolean	Deletes the file or directory represented by this File object. The method returns true if the deletion succeeds.
+renameTo(dest: File): boolean	Renames the file or directory represented by this File object to the specified name represented in dest. The method returns true if the operation succeeds.
+mkdir(): boolean	Creates a directory represented in this File object. Returns true if the directory is created successfully.
+mkdirs(): boolean	Same as mkdir() except that it creates directory along with it parent directories if the parent directories do not exist.

File Class

Obtaining file properties and manipulating file



Class java.io.File

The class java.io.File can represent either a file or a directory.

A **path string** is used to locate a file or a directory. Unfortunately, path strings are system dependent, e.g., **"c:\myproject\java\Hello.java"** in Windows or **"/myproject/java/Hello.java"** in Unix/Mac.

- Windows use back-slash '\' as the directory separator; while Unixes/Mac use forward-slash '/'.
- Windows use semi-colon ';' as path separator to separate a list of paths; while Unixes/Mac use colon ':'.
- Windows use "\r\n" as line delimiter for text file; while Unixes use "\n" and Mac uses "\r". **[ASCII 10(\n)/13(\r)]**
- The "c:\" or "\" is called the root. Windows supports multiple roots, each maps to a drive (e.g., "c:", "d:"). Unixes/Mac has a single root ("").



File Path

A path could be **absolute** (beginning from the root) or **relative** (which is relative to a reference directory). Special notations "." and ".." denote the current directory and the parent directory, respectively.

The `java.io.File` class maintains these system-dependent properties, for you to write programs that are portable:

- **Directory Separator:** in static fields `File.separator` (as String) and `File.separatorChar`. [They failed to follow the Java naming convention for constants adopted since JDK 1.2.] As mentioned, Windows use backslash '\'; while Unixes/Mac use forward slash '/'.
- **Path Separator:** in static fields `File.pathSeparator` (as String) and `File.pathSeparatorChar`. As mentioned, Windows use semi-colon ';' to separate a list of paths; while Unixes/Mac use colon ':'.



File Access Using URL Locator

You can construct a `File` instance with a path string or URI, as follows. Take note that the physical file/directory may or may not exist. A file URL takes the form of `file:///...`, e.g., `file:///d:/docs/programming/java/test.html`.

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child) // Constructs a File instance based on the
given path string.
public File(URI uri) // Constructs a File instance by converting from the given
file-URI "file:///..."
```

For examples,

```
File file = new File("in.txt"); // A file relative to the current working
directory
File file = new File("d:\\myproject\\java\\Hello.java");
// A file with absolute path
File dir = new File("c:\\temp"); // A directory
```

For applications that you intend to distribute as JAR files, you should use the `URL` class to reference the resources, as it can reference disk files as well as JAR'ed files , for example,

```
java.net.URL url = this.getClass().getResource("icon.png");
```



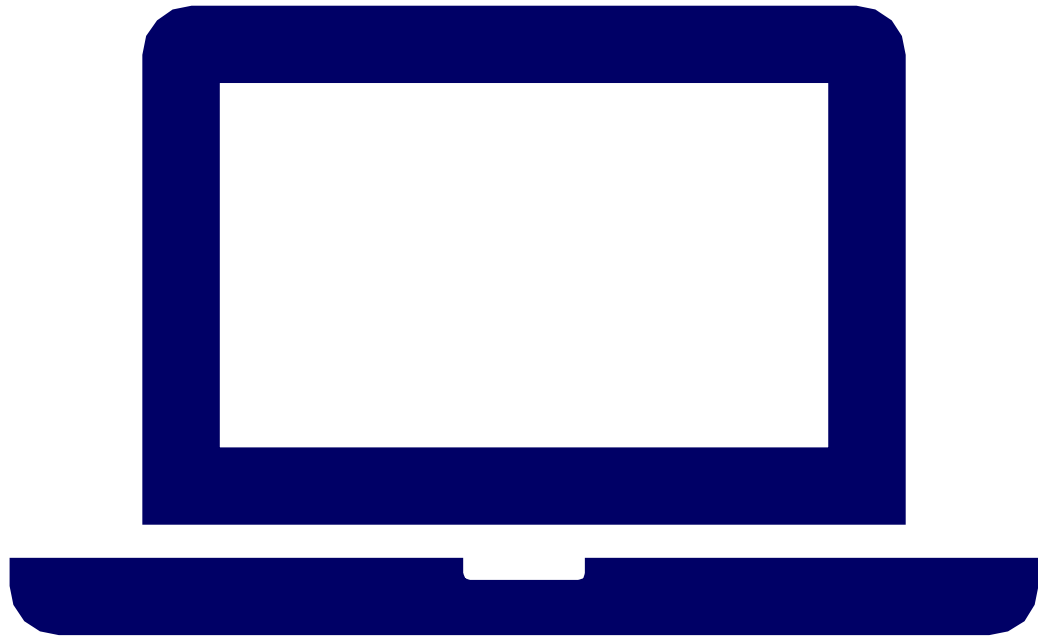
Methods to Verify a File/Directory and Lists Directory Files

Verify a File/Directory:

```
public boolean exists()           // Tests if this file/directory exists.
public long length()              // Returns the length of this file.
public boolean isDirectory()     // Tests if this instance is a directory.
public boolean isFile()          // Tests if this instance is a file.
public boolean canRead()         // Tests if this file is readable.
public boolean canWrite()        // Tests if this file is writable.
public boolean delete()          // Deletes this file/directory.
public void deleteOnExit()       // Deletes this file/directory when the program terminates.
public boolean renameTo(File dest) // Renames this file.
public boolean mkdir()           // Makes (Creates) this directory.
```

List Directory:

```
public String[] list()           // List the contents of this directory in a String-array
public File[] listFiles()        // List the contents of this directory in a File-array
```



Demonstration Program

LISTDIRECTORYRECURSIVE.JAVA



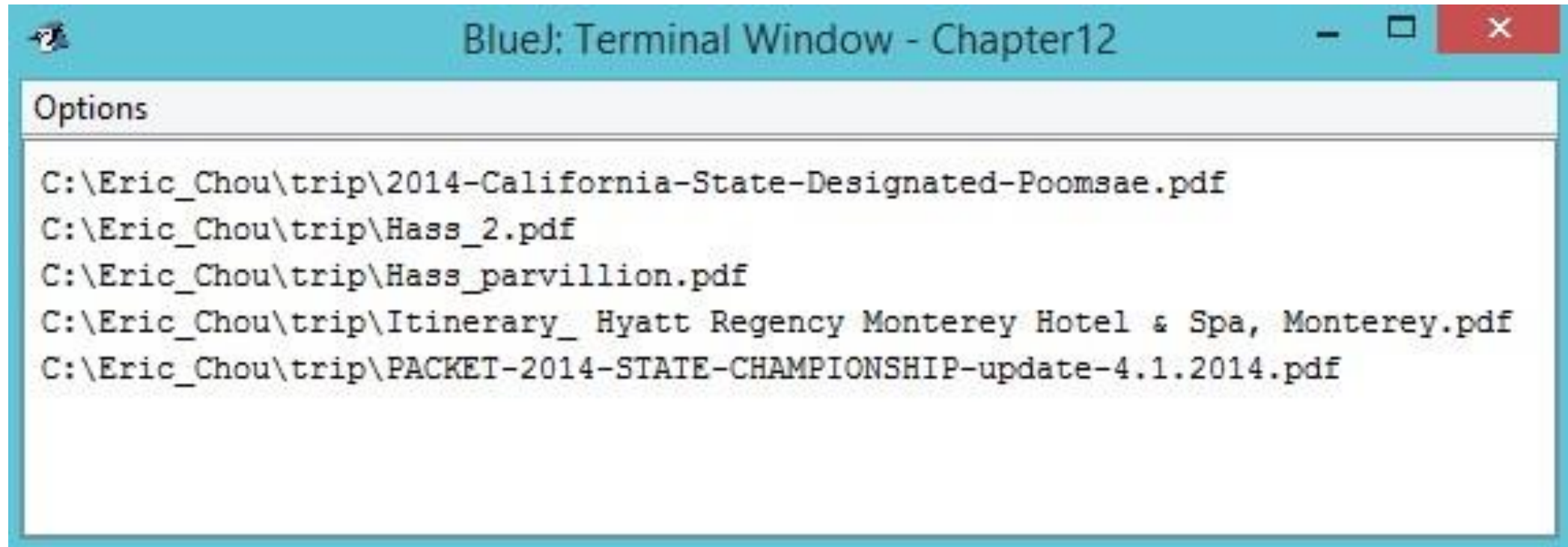
Demo Program:

The following program recursively lists the contents of a given directory (similar to Unix's "ls -r" command).

```
10 public class ListDirectoryRecursive {
11     public static void main(String[] args) {
12         File dir = new File("C:\\\\Eric_Chou\\trip"); // Escape sequence needed for '\\'; Try your own directory
13         listRecursive(dir);
14     }
15
16     public static void listRecursive(File dir) {
17         if (dir.isDirectory()) {
18             File[] items = dir.listFiles();
19             for (File item : items) {
20                 System.out.println(item.getAbsolutePath());
21                 if (item.isDirectory()) listRecursive(item); // Recursive call
22             }
23         }
24     }
25 }
```



Results:



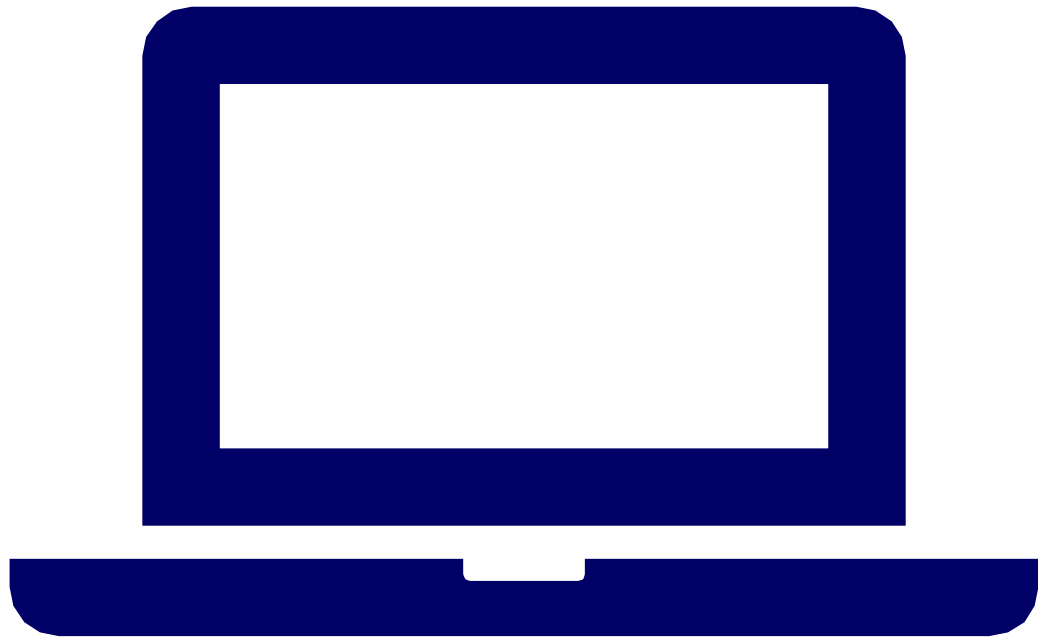
```
Options
C:\Eric_Chou\trip\2014-California-State-Designated-Poomsae.pdf
C:\Eric_Chou\trip\Hass_2.pdf
C:\Eric_Chou\trip\Hass_parvillion.pdf
C:\Eric_Chou\trip\Itinerary_ Hyatt Regency Monterey Hotel & Spa, Monterey.pdf
C:\Eric_Chou\trip\PACKET-2014-STATE-CHAMPIONSHIP-update-4.1.2014.pdf
```



Demo Program: Explore File Properties

`TestFileClass.java`

Use a program that demonstrates how to create files in a platform-independent way and use the methods in the File class to obtain their properties.



Demonstration Program

TESTFILECLASS.JAVA



TestFileClass

```
BlueJ: Terminal Window - Chapter12
Options
Does it exist? true
The file has 2462 bytes
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
Absolute path is C:\Eric_Chou\Udemy\APCSB\BlueJ\Chapter12\image\us.gif
Last modified on Wed Aug 05 15:25:47 PDT 2015
```

```
public class TestFileClass {
    public static void main(String[] args) {
        java.io.File file = new java.io.File("image/us.gif");
        System.out.println("Does it exist? " + file.exists());
        System.out.println("The file has " + file.length() + " bytes");
        System.out.println("Can it be read? " + file.canRead());
        System.out.println("Can it be written? " + file.canWrite());
        System.out.println("Is it a directory? " + file.isDirectory());
        System.out.println("Is it a file? " + file.isFile());
        System.out.println("Is it absolute? " + file.isAbsolute());
        System.out.println("Is it hidden? " + file.isHidden());
        System.out.println("Absolute path is " +
            file.getAbsolutePath());
        System.out.println("Last modified on " +
            new java.util.Date(file.lastModified()));
    }
}
```



Input Validation

LECTURE 8



Input Validation – “All Input is Evil”

- Any input that comes into a program from an external source – such as a user typing at a keyboard or a network connection – can be the source of security concerns and disastrous bugs. All input should be treated as potentially dangerous.



Risk – How Can It Happen?

- All input data is a potential source of problems. If input is not checked to verify that it has the correct type, format, and length, it can cause problems.
- Failure to validate input can lead to serious security risks such as integer error, buffer overflow, and SQL injections among others.



Example of Occurrence:

- A Norwegian woman mistyped her account number on an internet banking system. Instead of typing her 11-digit account number, she accidentally typed an extra digit, for a total of 12 numbers. The system discarded the extra digit, and transferred \$100,000 to the (incorrect) account given by the 11 remaining numbers.
- A simple dialog box informing her that she had typed too many digits may have avoided this expensive error.

```
1 import java.util.Scanner;
2 public class InputValidationExample {
3
4     public static void main(String[] args) {
5         int[] vals = new int[10];
6
7         for (int i = 0; i < 10; i++) {
8             vals[i] = (i+1)*(i+1);
9         }
10
11         System.out.print("Please type a number: ");
12         Scanner sc = new Scanner(System.in);
13         int which = sc.nextInt();
14
15         int square = vals[which-1];
16         System.out.println("The square of "+which+" is "+square);
17     }
18 }
```



Example

- This program has two input validation problems. The first problem occurs if the user inputs a non-integer value. In Java, this causes a **NumberFormatException** to be thrown. The second problem occurs if the user enters a value that does not lie between 0 and 9. In Java, this will lead to an **ArrayIndexOutOfBoundsException**.
- A robust program would catch this error, provide a clear and appropriate error message, and ask the user to re-type their input.



How can I properly validate input?

- Check your input: The basic rule is for input validation is to check that input data matches all of the constraints that it must meet to be used correctly in the given circumstance. In many cases, this can be very difficult: confirming that a set of digits is, in fact, a telephone number may require consideration of the many differing phone number formats used by countries around the world.
- Some of the checks that you might want to use include:



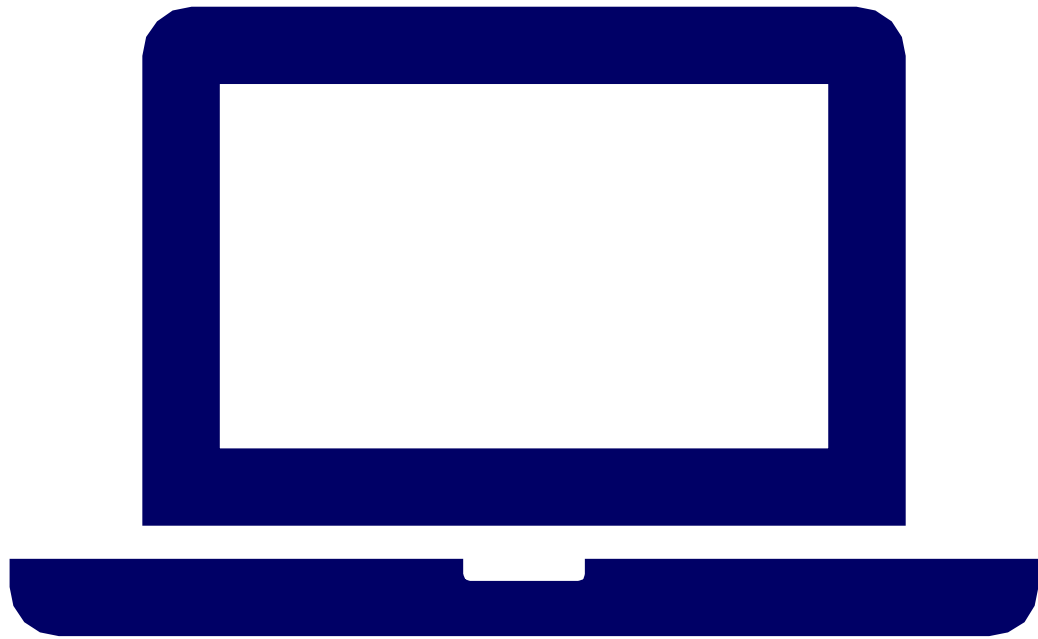
How can I properly validate input?

- **Type:** Input data should be of the right type. Names should generally be alphabetic, numbers numeric. Punctuation and other uncommon characters are particularly troubling, as they can often be used to form the basis of code-injection attacks. Many programs will handle input data by assuming that all input is of string form, verifying that the string contains appropriate characters, and then converting the string into the desired data type.
- **Range:** Verify that numbers are within a range of possible values: For example, the month of a person's date of birth should lie between 1 and 12. Another common range check involves values that may lead to division by zero errors.
- **Plausibility:** Check that values make sense: a person's age shouldn't be less than 0 or more than 150.



How can I properly validate input?

- **Presence check:** Guarantee presence of important data – the omission of important data can be seen as an input validation error.
- **Length:** Input that is either too long or too short will not be legitimate. Phone numbers generally don't have 39 digits; Social Security Numbers have exactly 9
- **Format:** Dates, credit card numbers, and other data types have limitations on the number of digits and any other characters used for separation. For example, dates are usually specified by 2 digits for the month, one or two for the day, and either two or four for the year.
- **Checksums:** Identification numbers such as bank accounts, often have check digits: additional digits included at the end of a number to provide a verifiability check. The check digit is determined by a calculation based on the remaining digits – if the check digit does not match the results of the calculation, either the ID is bad or the check digit is bad. In either case, the number should be rejected as invalid.



Demonstration Program

INPUTVALIDATION.JAVA

INPUTVALIDATIONEXAMPLE.JAVA



File Validation

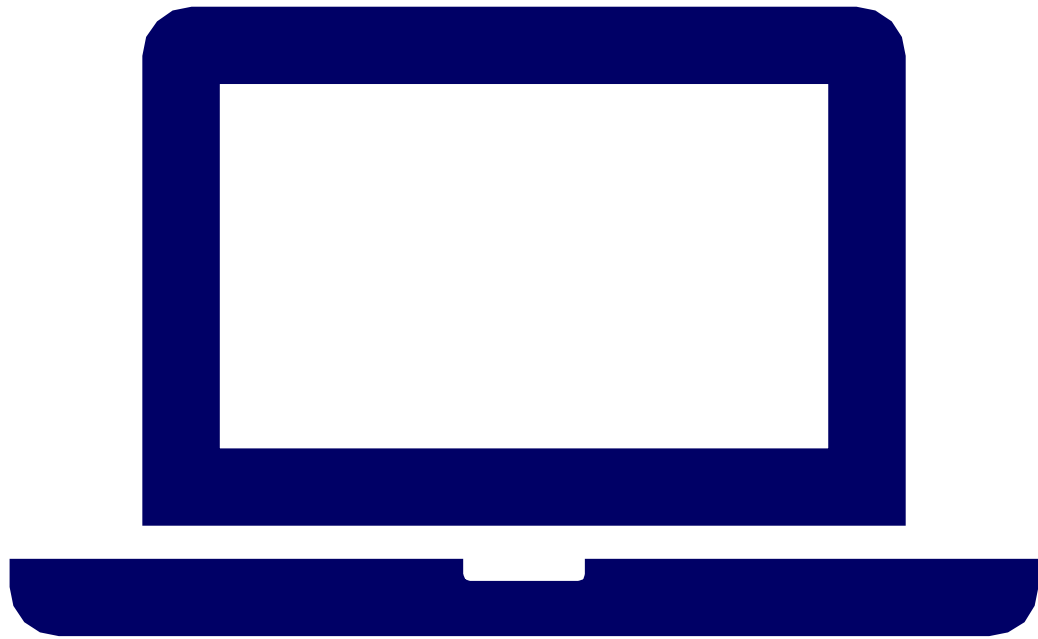
LECTURE 9



File Validation

- Check if the file opening operation is successfully executed.
- If not, the program will loop back because an exception will be thrown.
- If yes, then, the program will continue.

```
1 import java.io.*;
2 import java.util.*;
3 public class FileValidation{
4     public static void main(String[] args){
5         System.out.print("\f");
6         boolean done = false;
7         File f;
8         String fn = "";
9         Scanner in = new Scanner(System.in);
10        Scanner inf = null;
11        do{
12            try{
13                System.out.print("Enter a File Name: ");
14                fn = in.nextLine().trim();
15                inf = new Scanner(new File(fn));
16                done = true;
17            }
18            catch (Exception e){
19                System.out.println("File Missing");
20            }
21        } while (!done);
22
23        while (inf.hasNext()){
24            String s = inf.next();
25            System.out.println(s);
26        }
27        in.close();
28        inf.close();
29    }
30 }
```



Demonstration Program

INPUTVALIDATION.JAVA

INPUTVALIDATIONEXAMPLE.JAVA