Exception
Assertion and Text I/O

Shing-Chi Cheung
Computer Science and Engineering
HKUST

http://tutorials.jenkov.com/java-exception-handling/index.html

Part I: Exception

Motivation

- **Runtime errors** refer to errors that occur at execution
- Examples are:
  - Memory exhaustion
  - Files cannot be opened
  - Division by zero
  - Dereferencing a null value
- Execution **terminates abnormally** when runtime errors occur
- **Can we program the handling of runtime errors so that software continues or terminates gracefully when errors occur**

Motivation

Consider a program that reads teaching members of COMP3021.

```
public class Motivation {
  String filename;

  public Motivation() {
    // Ask users for the input filename
    System.out.print("Input Filename: ");
    Scanner sc = new Scanner(System.in);
    filename = sc.nextLine();
  }
```

```
public void readMemberNames() {
  // Create a File and a Scanner object
  File inputFile = new File(filename);
  Scanner sc = new Scanner(inputFile);
  String[] members = new String[3];

  // Read the content using a loop
  for(var i=0; i<members.length; i++) {
    String name = sc.nextLine();
    members[i] = name;
  }

  // Close the file and compute output
  sc.close()
  …
}
```

Motivation – Potential Runtime Errors

Program Statements	Possible Factors Causing Runtime Errors
filename = sc.nextLine();	No line is entered or System.in buffer is not ready
File inputFile = **new** File(**filename**);	filename contains a null path name
Scanner sc = **new** Scanner(inputFile);	inputFile is a null value Refers to an non-existent file
String name = sc.nextLine();	No more line is found This scanner has already been closed by its environment

- Many statements are subject to runtime errors induced by factors external to program logic.
- It is difficult for programmers to predict the occurrences of all such factors in advance when coding.

Prevent program from crashing due to runtime errors

Program Statements	Possible Runtime Errors
Scanner sc = **new** Scanner(System.***in***);	System.in is misconfigured to a null value
filename = sc.nextLine();	No line is entered or System.in buffer is not ready
File inputFile = **new** File(**filename**);	filename contains a null path name
Scanner sc = **new** Scanner(inputFile);	inputFile is a null value Refers to an non-existent file
String name = sc.nextLine();	No line is found This scanner has already been closed by its environment
sc.close()	This scanner has already been closed by its environment

Brute-force solution:

- Add an if-statement before each statement to check against the situations that may induce runtime errors
  - Adding such if-statements is tedious and error-prone
  - Reduce code maintainability
  - The added if-statements obstruct code understanding

Brute-Force Solution

☞ *add 6 if-statements*

```
public class Motivation {
  String filename;

  public Motivation() {
    // Ask users for the input filename
    System.out.print("Input Filename: ");
    Scanner sc = new Scanner(System.in);
    filename = sc.nextLine();
  }
```

```
public void readMemberNames() {
  // Create a File and a Scanner object
  File inputFile = new File(filename);
  Scanner sc = new Scanner(inputFile);
  String[] members = new String[3];

  // Read the content using a loop
  for(var i=0; i<members.length; i++) {
    String name = sc.nextLine();
    members[i] = name;
  }

  // Close the file and compute output
  sc.close()
  …
}
```

Exception Handling

Program Statements	Possible Runtime Errors
Scanner sc = **new** Scanner(System.***in***);	System.in is misconfigured to a null value
filename = sc.nextLine();	No line is entered or System.in buffer is not ready
File inputFile = **new** File(**filename**);	filename contains a null path name
Scanner sc = **new** Scanner(inputFile);	inputFile is a null value Refers to an non-existent file
String name = sc.nextLine();	No line is found This scanner has already been closed by its environment
sc.close()	This scanner has already been closed by its environment

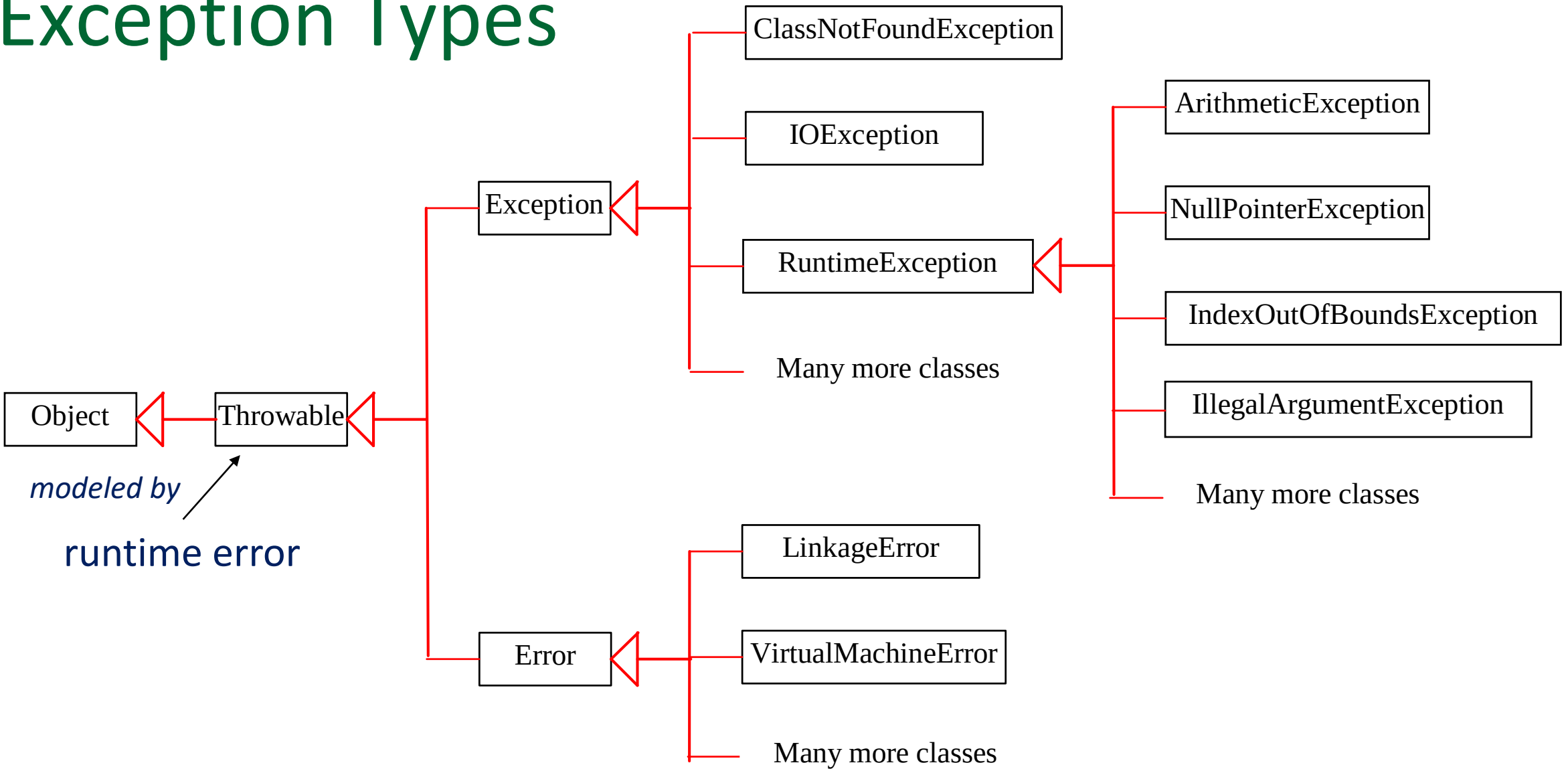
Better solution:

- Handle these situations altogether using a try-catch block.

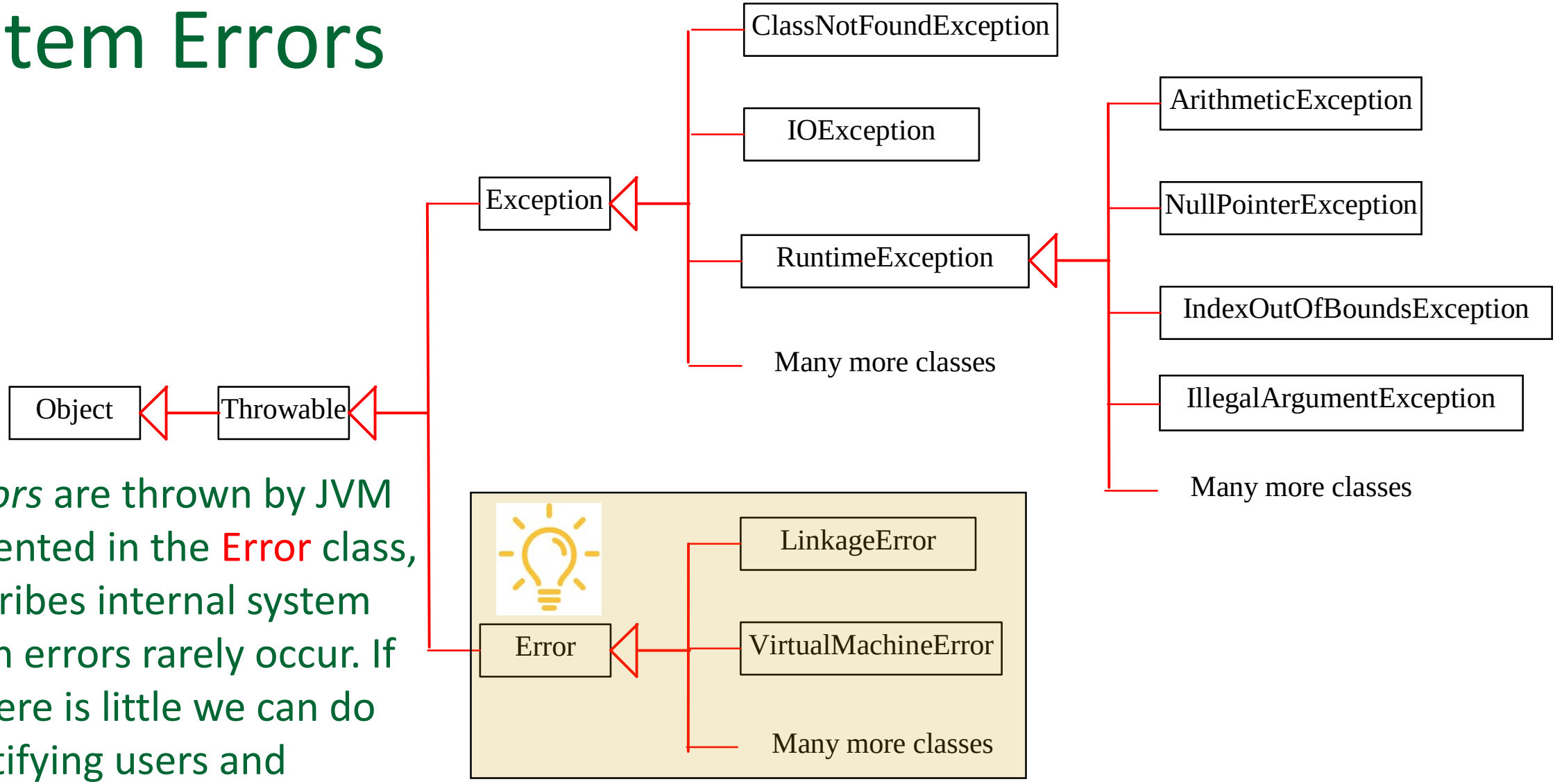
```
Motivation demo = null;
try {
  demo = new Motivation();
  demo.readMemberNames();
} catch(Exception e) {
  System.out.println("Please check if " + demo.filename + " is a valid non-empty text file!");
}
```

Motivation.java

Exception Types



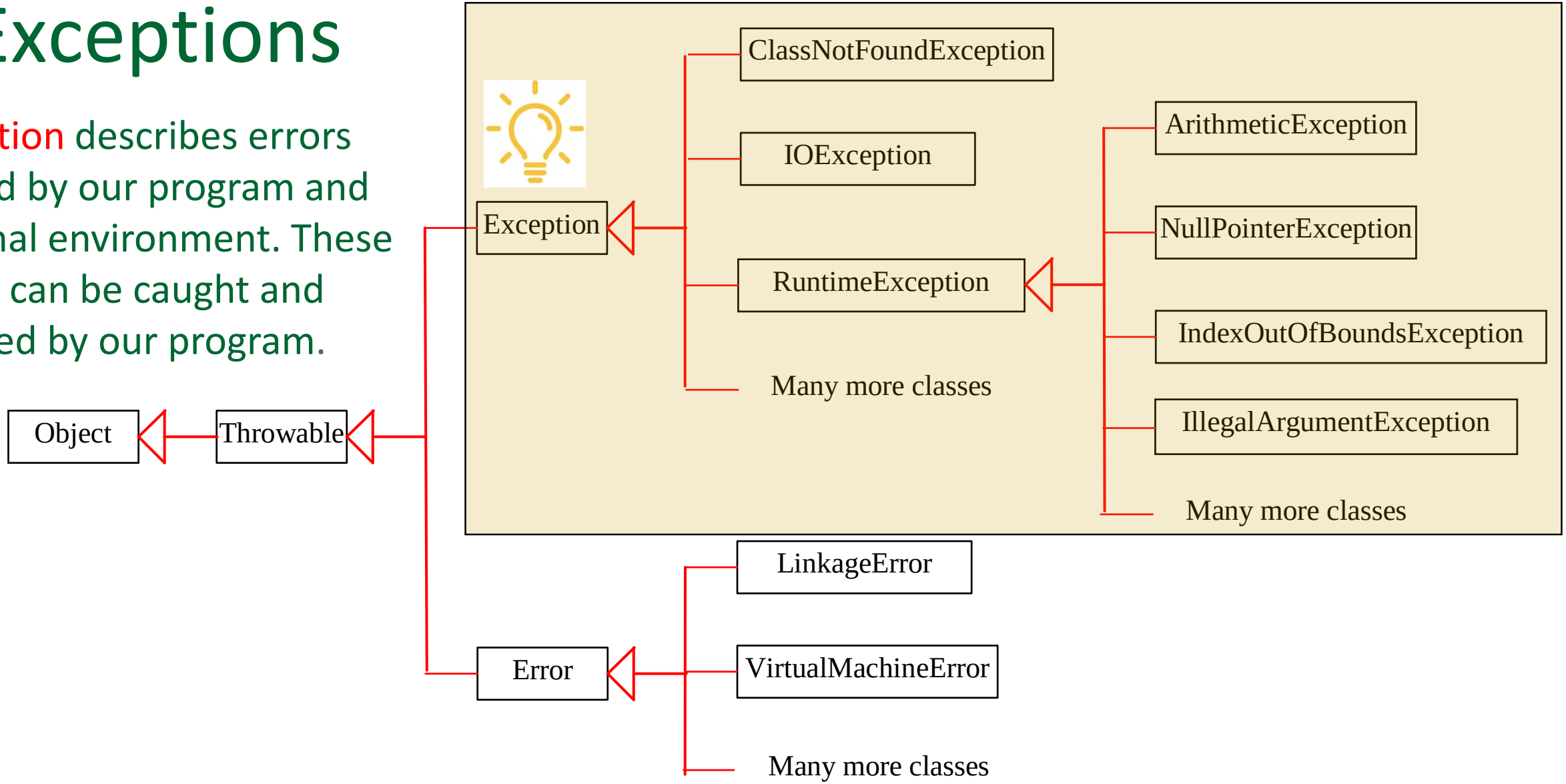
System Errors



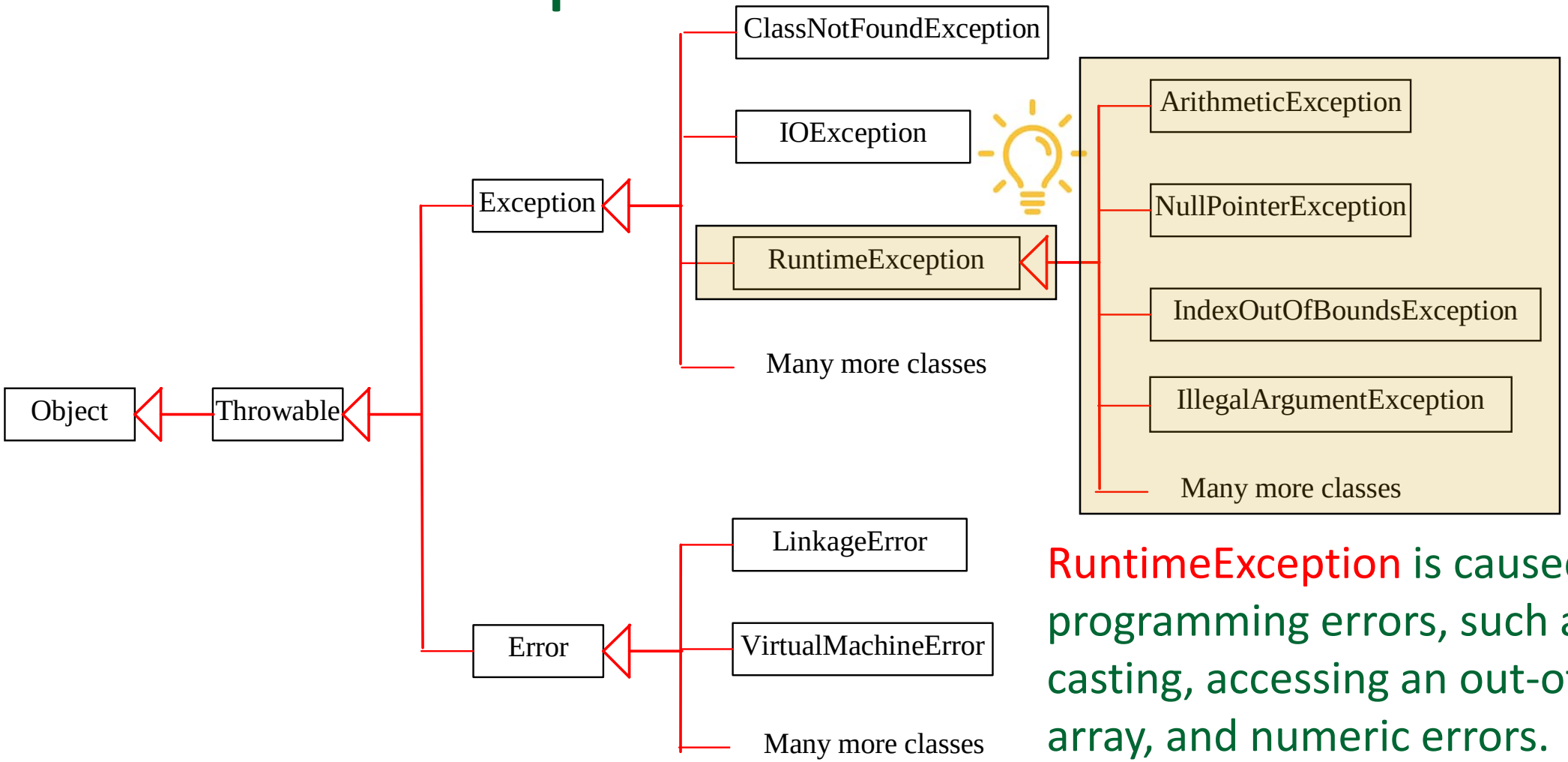
System errors are thrown by JVM and represented in the Error class, which describes internal system errors. Such errors rarely occur. If they do, there is little we can do beyond notifying users and terminating the program gracefully.

Exceptions

Exception describes errors caused by our program and external environment. These errors can be caught and handled by our program.



Runtime Exceptions



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

Note: RuntimeException models only common programming errors.

Commonly Used Built-in Exceptions

Built-in Exception Class	Description
ArithmeticException	Division by zero or some kinds of arithmetic errors
ArrayIndexOutOfBoundsException	An array index is less than zero or greater than or equal to array's length
IllegalArgumentException	Improper actual parameter is used when calling a method
NullPointerException	Dereference a null value
NumberFormatException	Illegal number format is used
IOException	Errors found in file i/o
CloneNotSupportedException	Class is not cloneable

unchecked exceptions

checked exceptions

The Exception Class

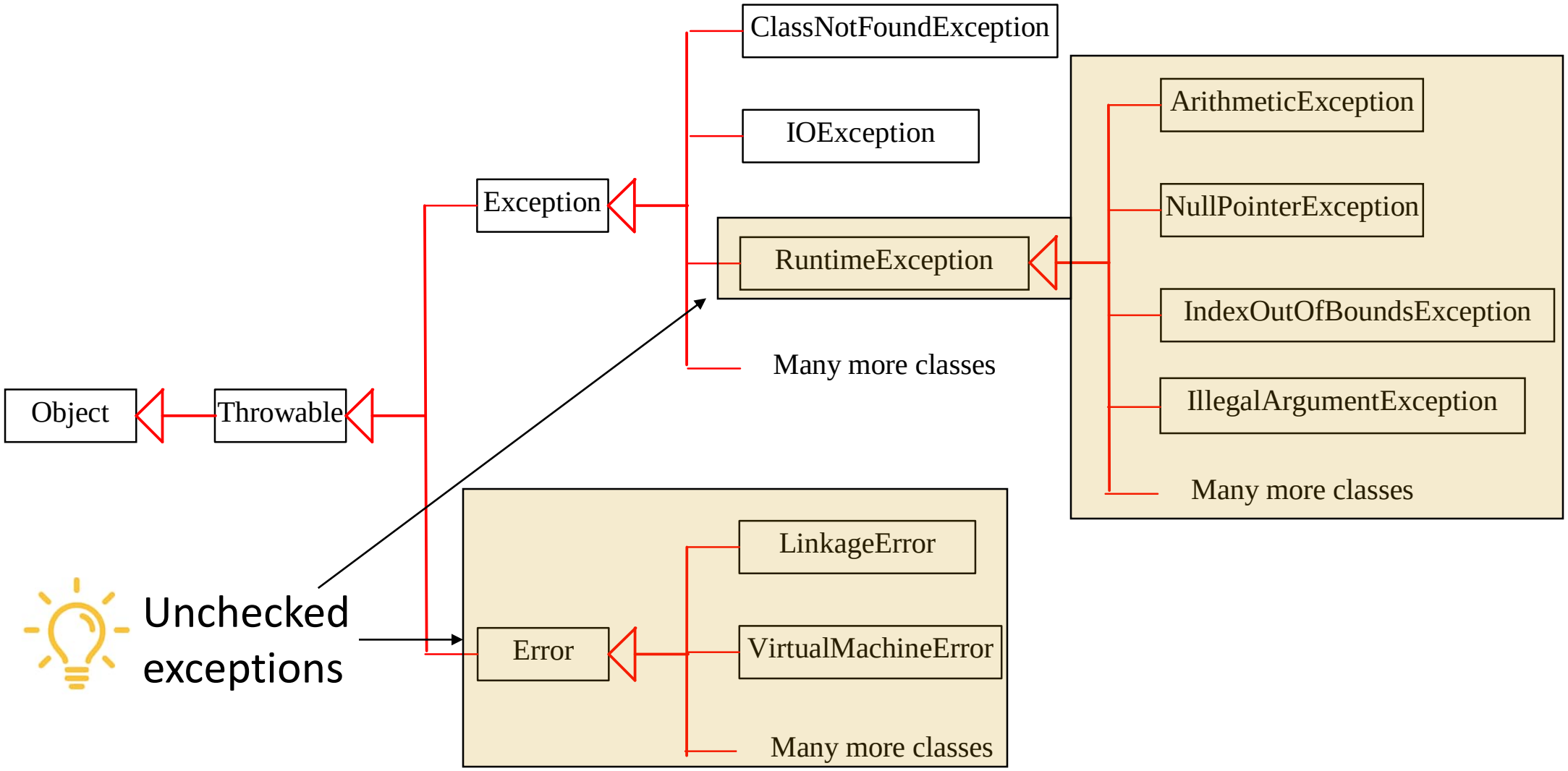
- Exception is the **root class** of all exceptions
- It has two constructors
  - **Exception()**
  - **Exception(String message)**
  - Example: **throw new** Exception("Invalid data"); *// throw an exception to its caller*
- It provides methods to retrieve diagnosis information when an exception occurs
  - **getMessage()**: returns the message of the exception object
  - **toString()**: returns a short description of the exception object
  - **printStackTrace()**: print the trace of all the methods that were being called when the exception occurred

Motivation.java *// demo diagnosis information*

Two Exception Types: Unchecked vs Checked

- There are two major types of exceptions that can be thrown while program execution
- Unchecked exceptions
  - RuntimeException, Error and their subclasses are known as *unchecked exceptions*
- Checked exceptions
  - All exceptions other than the unchecked ones above

Unchecked Exceptions



Checked Exceptions

- When using a statement that can throw a checked exception, compiler checks if the exception will be caught
- A checked exception can be caught using a try/catch block
  - Example:

```
try {
   … // statements that can throw checked exceptions, say, of class E1 and E2
} catch (E1 e) {
   … // statements that handle an E1 exception
} catch (E2 e) {
   … // statements that handle an E2 exception
}
```

Checked Exceptions (cont.)

- A method may also leave the catching of a checked exception to its caller when it is unclear how to handle it at the current method (using the throws clause)

Example:

```
public void readMemberNames() throws FileNotFoundException, NoSuchElementException {
  // Create a File and a Scanner object
  var inputFile = new File(filename);
  var sc = new Scanner(inputFile); // can throw a checked exception
  var members = new String[3];
  …
}
```

Motivation.java

Unchecked Exceptions

- In most cases, exceptions arising from RuntimeException reflects programming logic errors that are not recoverable. Examples:
  - NullPointerException
  - IllegalArgumentException
  - IndexOutOfBoundsException
- These logic errors should be corrected in the program logic.
- Java compiler does not check if our program catches RuntimeException
  - Because if we know our program contains logic errors, we should have already corrected them

Unchecked Exceptions

- Error objects arising from the Error class mostly reflect issues in the Java virtual machine. Examples:
  - LinkageError
  - VirtualMachineError

- There is little we can do when these errors arise.
- Java compiler does not check if our program catches Error objects

Note: AssertionError is a subclass of Error but it is unrelated to JVM issues. We will discuss it later in Part II

Declaring Exceptions

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

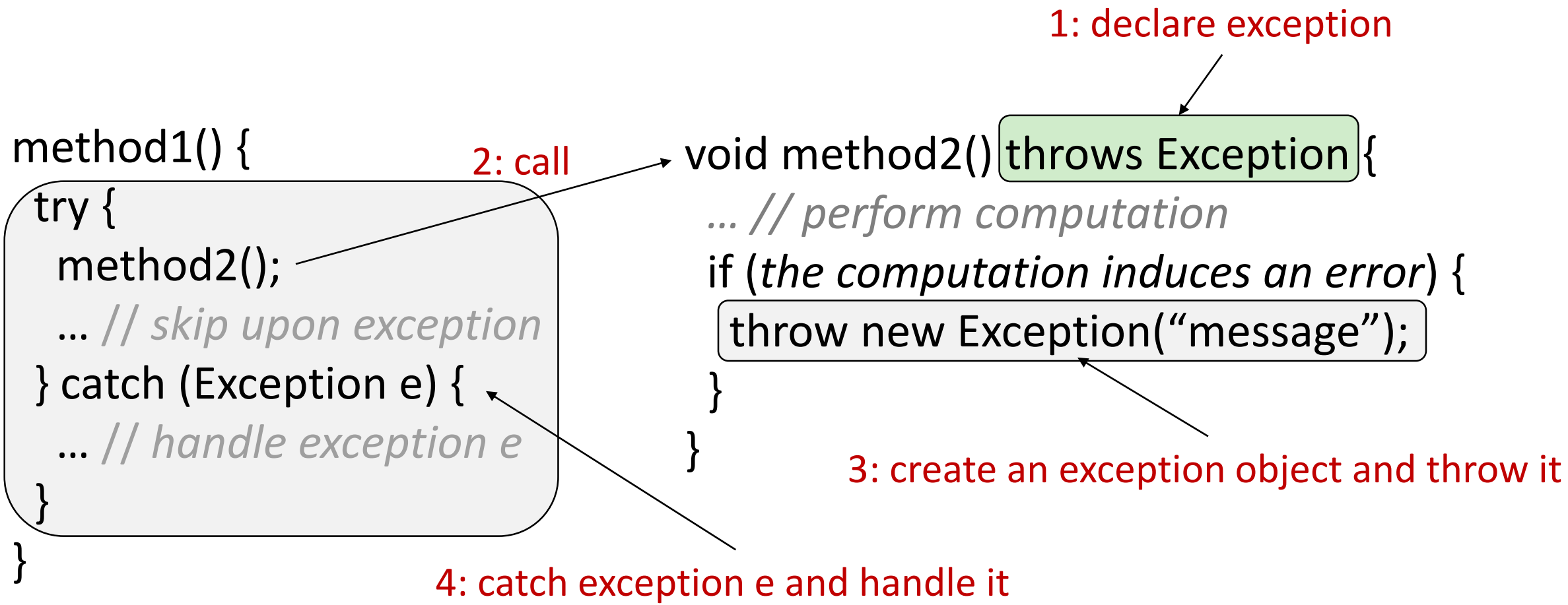
```
public void myMethod() throws IOException { … }
public void myMethod()
    throws IOException, AnOtherException { … }
```

Guidelines for Checked and Unchecked Exceptions

- If an exception thrown is an instance of RuntimeException or Error, it is an unchecked exception
  - We can optionally catch it and/or declare it
- Otherwise, it is a checked exception. We have two alternatives:

  - catch it within the method where the exception is thrown using a try-catch block, and/or
  - declare it: throws the exception at the method declaration

Declaring, Throwing and Catching Exceptions



Declaring Exceptions

- Every method must declare the types of checked exceptions that it can throw
- This is known as declaring exceptions

Syntax

```
<access modifier> <return type> <method name>(<parameter list>)
  throws <exception class name 1>, ..., <exception class name n>
{
  // ...
}
```

where <access modifier> is a Java keyword to control the access to the method, <return type> is the type name of the return value, <method name> is the name of the method, <parameter list> is a list of formal parameters of method, <exception class name 1>, ..., <exception class n> are list of exception class names

Throwing Exceptions

- When an error is detected, a program creates an instance of an appropriate exception type and throw it.
- This is known as throwing an exception

Syntax

```
throw new <exception class name>();

<exception class name> <variable name> =  new <exception class name>();
throw <variable name>;
```

where <exception class name> is the name of an exception class, <variable name> is the name of a reference variable that references to an exception class object

Throwing Unchecked Exceptions Example

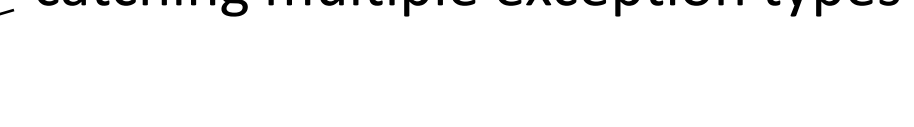
```
/** Set a new radius */
public void setRadius(double newRadius) throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException("Radius cannot be negative");
}
```

a subclass of RuntimeException

UnCheckedExceptionExample.java

Catching Multiple Exceptions

```
try {
    statements;  // Statements that may throw exceptions
} catch (Exception1 exVar1) {
    handler for exception1;
} catch (Exception2 exVar2) {
    handler for exception2;
} catch (Exception3 exVar3 | Exception4 exVar4) {
    handler for exception3 and exception4;
}
```



MotivationMultipleExceptions.java

Cautions in Using Multiple Exceptions

- Are there any errors with the following code? Why?

```
try {…}
catch (Exception e1) {…}
catch (ArithmeticException e2) {…}
catch (IOException e3) {…}
```

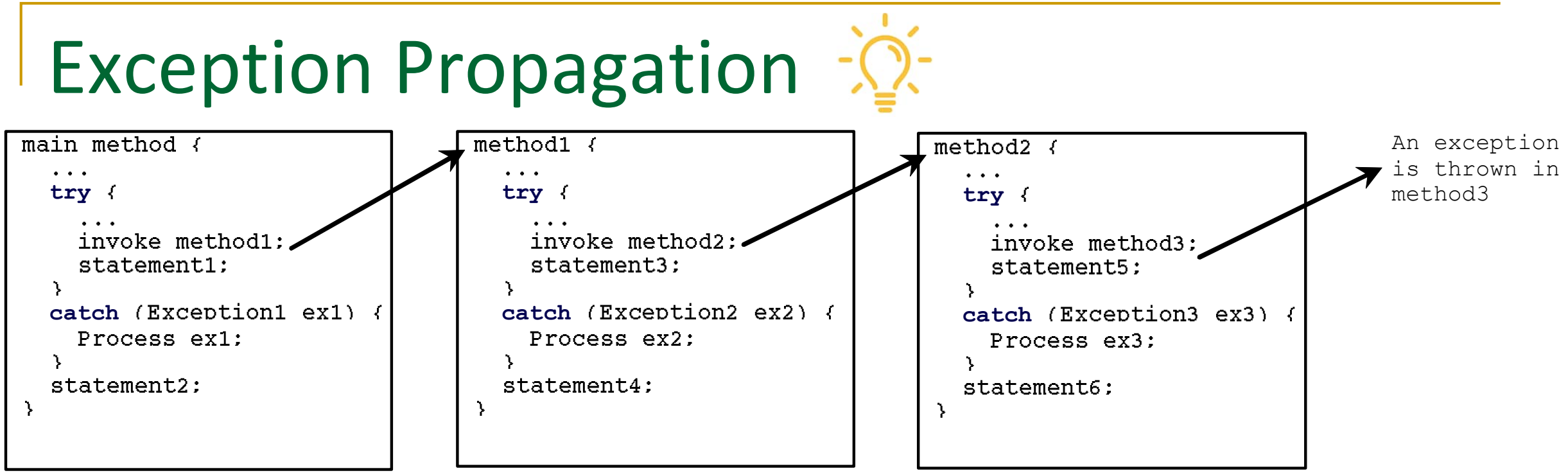
Cautions in Using Multiple Exceptions

- Are there any errors with the following code? Why?

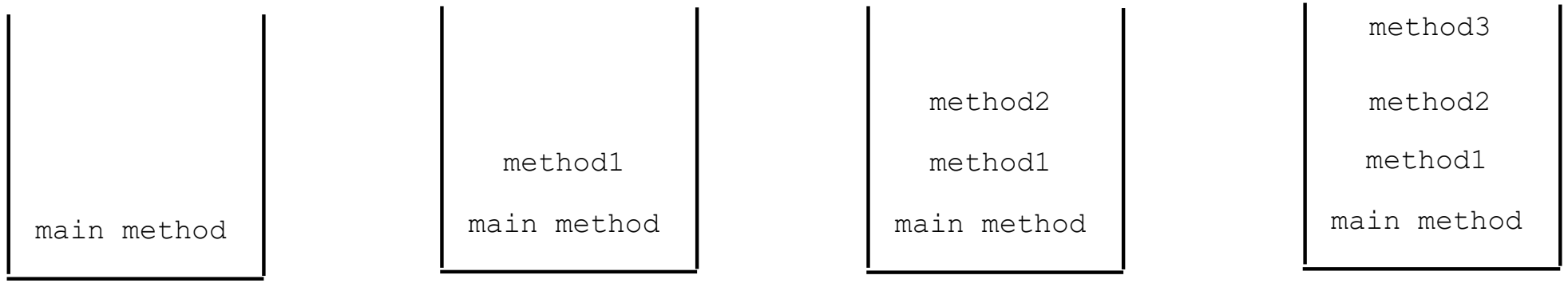
```
try {…}
catch (Exception e1) {…} // Move it to the bottom
catch (ArithmeticException e2) {…}
catch (IOException e3) {…}
```

Recommendation:
Put the catch blocks for a more specific exception before a more general one

Exception Propagation



Call Stack



Rethrowing Exceptions

```
try {
   statements;
}
catch (TheException ex) {
   perform operations before exits;
   throw ex;
}
```

Suggest a situation where we would like to rethrow an exception?

MotivationRethrow.java

The finally Clause

```
try {
  statements;
}
catch (TheException ex) {
  handle ex;
}
finally {  // executed on exiting the try-catch block
  finalStatements;
}
```

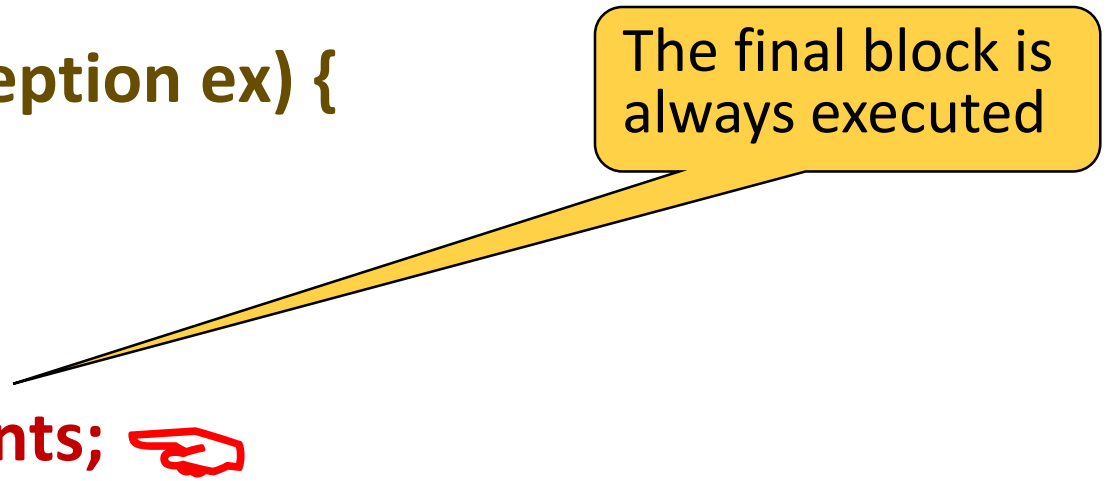
Trace a Program Execution - 1

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

Next statement;
```

Suppose no exceptions in the statements

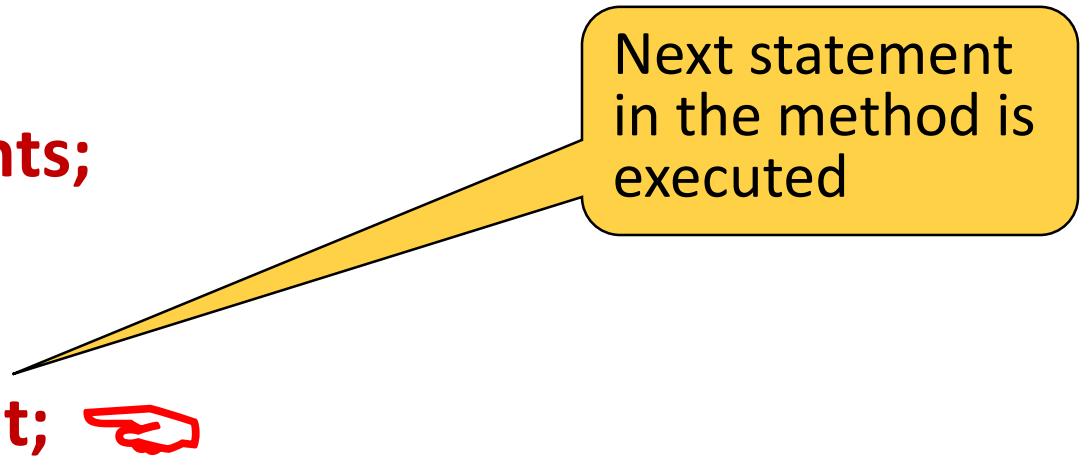
Trace a Program Execution - 1

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

Next statement;
```



Trace a Program Execution - 1

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

Next statement;
```



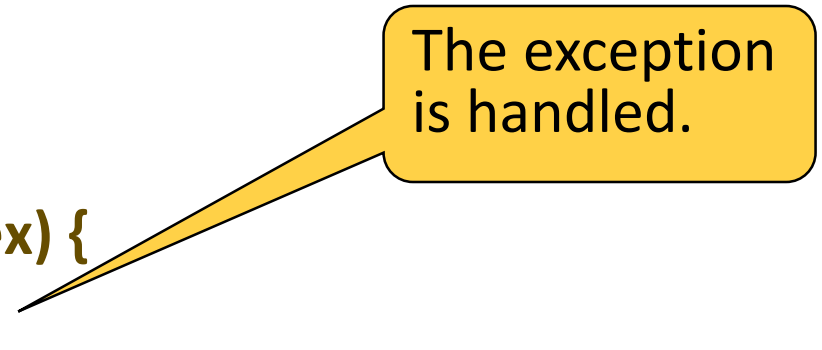
Trace a Program Execution - 2

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

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

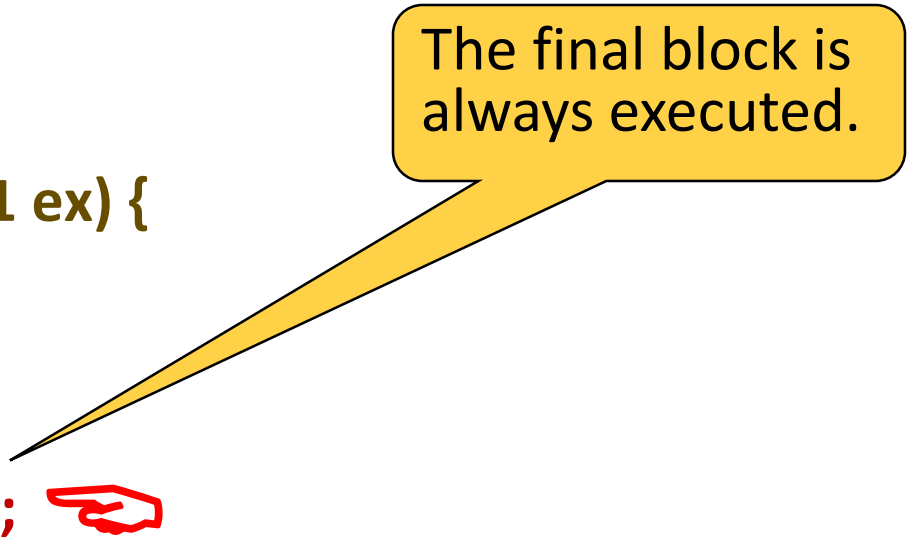
Trace a Program Execution - 2

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

Next statement;
```



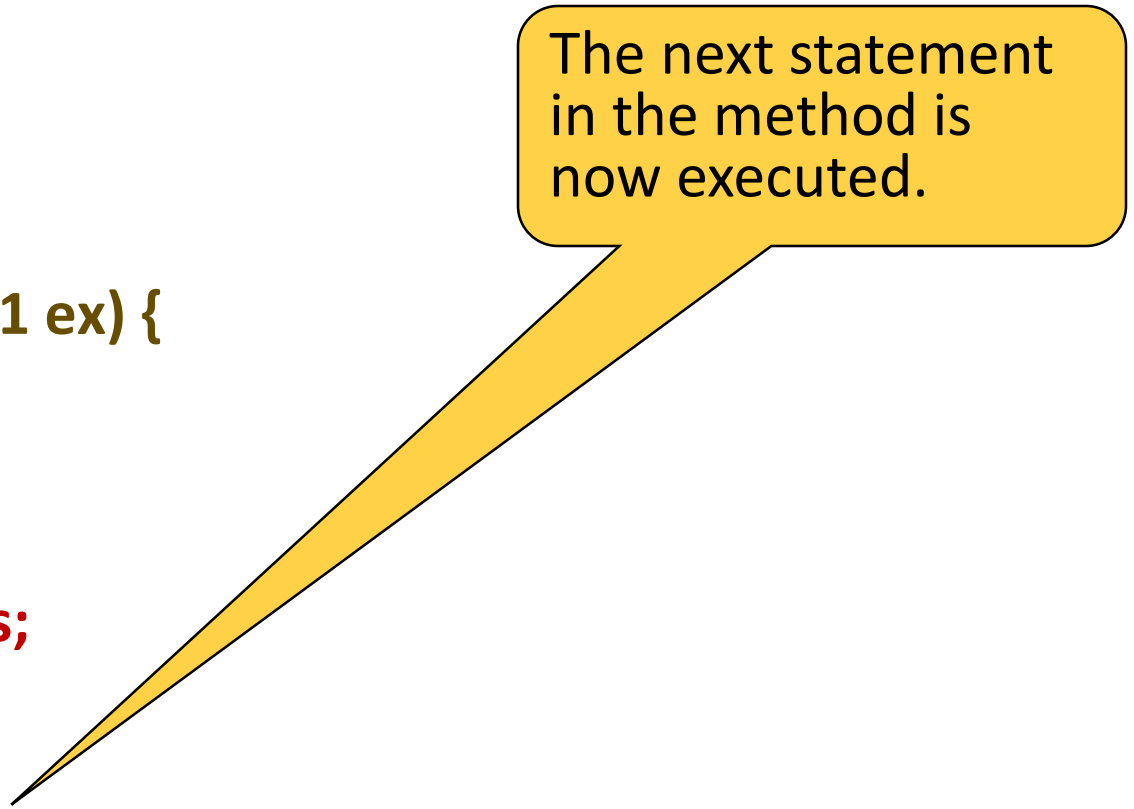
Trace a Program Execution - 2

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

Next statement;
```



Trace a Program Execution - 2

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

Next statement;
```



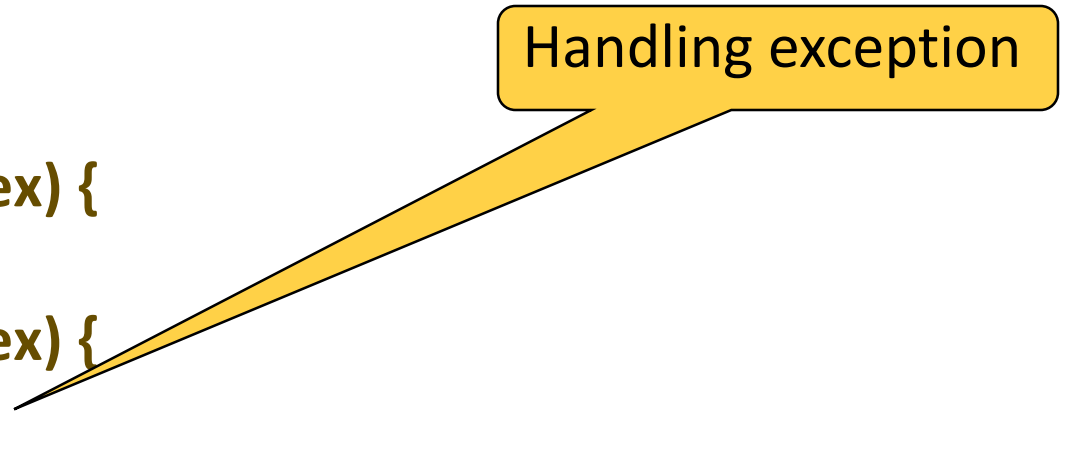
Trace a Program Execution - 3

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

Next statement;
```


statement2 throws an exception of type Exception2.

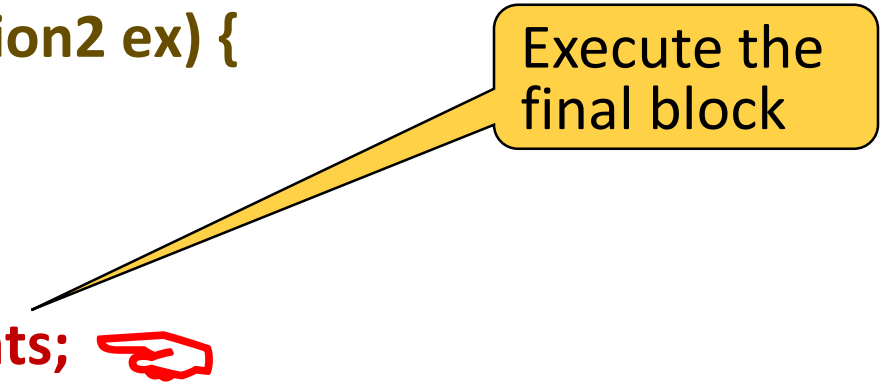
Trace a Program Execution - 3

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

Next statement;
```



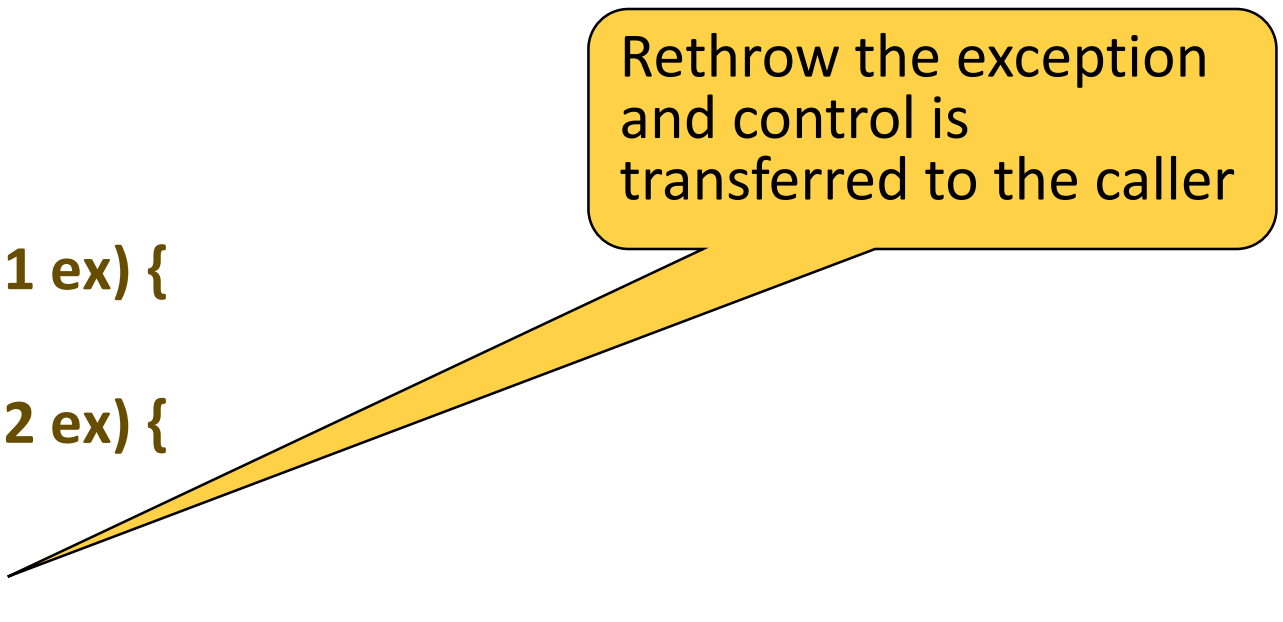
Trace a Program Execution - 3

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

Next statement;
```



Trace a Program Execution - 3

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

Next statement;
```



MotivationRethrowFinally.java

Try with Resources

```
Scanner sc = null;
try {
  …
  sc = new Scanner(inputFile);
  …
} catch (Exception e) {
  …
} finally {
  sc.close();
}
```

A problem with try-catch-finally:

- Variable sc is only used within the try-catch-finally block.
- However, we need to declare variable sc outside the block so that it can be accessed in try {}, catch{} and finally {}.
- We also need to redundantly initialize sc to null. Why?

Try with Resources

declare the resources to be used

```
try (var sc = new Scanner(inputFile)) {
  …
} catch (Exception e) {
   …
}
```

- Variable sc is accessible anywhere within the try-catch-finally block but not outside the block.
- All declared resources will be automatically closed upon leaving the block.
- There is no need to add sc.close() in the finally block.

Note: Separate multiple resources using ';': try (var res1 = …; var res2 = …) { … }
Closeable resources must implement java.lang.AutoCloseable interface.

MotivationRethrowResources.java
WriteDataWithAutoClose.java

Caution: Conflicting Interfaces

- A class may implement two interfaces with conflicting information
- Conflicts may arise from:
  - Two constants with the same name but different values
  - Two methods with the same signature but different return types
  - Two methods throwing different exceptions

TestInterfaceWithException.java

- The Java language is designed to detect these conflicts by compiler

Notes on Using Exceptions

- Exception handling separates error-handling code from normal programming tasks
- This makes programs easier to read and to modify
- Note that exception handling can require more computational resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.

Notes on Using Exceptions

As exception handling can require additional computational resources, we should use it to deal with **unexpected** error conditions. Do not use it to deal with simple, expected situations. For example:

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

✘

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

✓

ComputeAverageMarksV1.java

ComputeAverageMarksV2.java

Notes on Using Exceptions

- If a method has acquired resources (e.g., network connections, database connections and files), it should catch exception and release these acquired resources in the **finally** block
  Q: Why the finally block?
- A method throws an exception to its caller when it wants the exception to be processed by its caller
- If a method has the full capability to handle the exception, there is no need to throw an exception to its caller

Customized Exception Classes

- Use the built-in exception classes whenever possible.
- The built-in classes are insufficient if we want to **store additional information** specific to the unexpected errors.
  - For example, we want to store the radius of a circle when it triggers an error. This requires an additional field in the Exception object.
- Customize exception classes by extending Exception or a subclass of Exception.

```
public class InvalidRadiusException
   extends Exception {

  private double radius;

  public InvalidRadiusException(double radius) {
   super("Invalid radius " + radius);
   this.radius = radius;
  }
  public double getRadius() {
   return radius;
  }
}
```

InvalidRadiusException.java
TestCircleWithCustomizedException.java

From Head First Java

- You cannot have a catch or finally without a try

```
void go() {
  Foo f = new Foo();
  f.foof();
  catch(FooException ex) { }
}
```

ILLEGAL! Where's the try?

- You cannot put code between the try and the catch

```
try {
  x.doStuff();
}
int y = 43; // WRONG
catch(Exception ex) { }
```

ILLEGAL! You can't put code between the try and the catch

- A try MUST be followed by either a catch or a finally

```
try {
  x.doStuff();
} finally {
  // cleanup
}
```

LEGAL because you have a finally, even though there's no catch. But you cannot have a try by itself

- A try with only a finally (no catch) must still declare the exception

```
void go() throws FooException {
  try {
    x.doStuff(); // may throw FooException
  } finally { }
}
```

A try without a catch doesn't satisfy the handle or declare law

Part II: Assertion (Self-Study)

Assertions

```
assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
```

assumption that should hold

message displayed when assumption does not hold

- An assertion is a Java statement that tests an assumption after performing a computation task.
- An assertion contains a boolean expression that should be evaluated to true when it is executed.
- Assertions can be used to detect logic errors and enhance program correctness.

Assertions

- When an assertion is executed, its boolean expression is evaluated
- If it is false, an AssertionError object will be thrown
- The program displays a message on the console and exits.

Assertions can be written in two forms:

- assert <boolean-expression> ; *// no display message object is specified*
- assert <boolean-expression> : <detailed-message> ;

Executing Assertions

- When an assertion statement is executed, Java evaluates the assertion. If it is false, an AssertionError will be thrown.
- AssertionError is a subclass of Error.
- When an assertion becomes false, the program displays the specified message on the console and exits.

```
assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
```

Executing Assertions Example

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

Running Programs with Assertions

- By default, the assertions are disabled at runtime. To enable it, use the switch –ea as follows:

```
java –ea AssertionDemo
```

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

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

Running Programs with Assertions

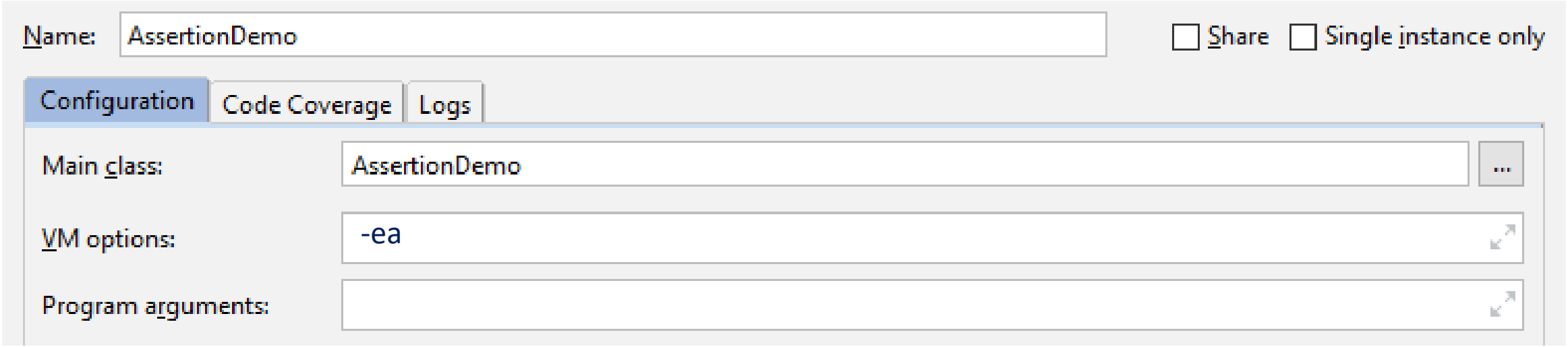
- Assertions are normally disabled
- Assertions are enabled only while testing or debugging a program

```
java –ea AssertionDemo
```

- Checks that should always be enabled are unlikely assertions.
- Assertions can be leveraged by automated tools for test generation and debugging

IntelliJ Configuration

- Run -> Edit Configurations… -> select AssertionDemo



Using Exception Handling or Assertions

- Exceptions deal with unexpected situations occurring in the runtime environment. They offer robustness.
- Assertions should not be used to replace exception handling.
- Assertions detect program logic errors and enhance program correctness. They offer validity checks.
- Assertions are checked at runtime and can be turned on or off before running the program.

Using Exception Handling or Assertions

- Do not use assertions for argument checking in public methods.
- Passing valid arguments to a public method is considered part of the method's contract, which must always be obeyed whether assertions are enabled or disabled.
  - @Nullable
  - @NotNull

@Nullable and @NotNull

Note: May not be supported by all Java IDEs so far

- Two useful annotations supported by IntelliJ for argument validation
  - @Nullable <type> <variable> *// The variable may take a null value*
  - @NotNull <type> <variable> *// The variable cannot take a null value*
  - Example: **boolean** withdraw(@NotNull Account acc, **double** amt) {…}
- IntelliJ will conduct a check whether a **null** value could be passed to the annotated parameter
- Checks can be configured in the Settings/Preferences

https://www.jetbrains.com/help/idea/nullable-and-notnull-annotations.html

https://docs.oracle.com/javaee/6/api/javax/validation/constraints/NotNull.html

NonNull.java

Using Assertions

- A common use of assertions is to 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;
}
```

Question: If-then-else vs Exception vs Assert

Given: **boolean** withdraw(Account acc, **double** amt) { … }

- Which mechanism should we use?
  - acc.bal >= amt
  - amt >= 0
  - acc != null
  - Fail to read from or write to database
  - Suppose b is the balance at the entry of withdraw and b’ is the balance at the exit of withdraw: b == b’ + amt

Question: If-then-else vs Exception vs Assertion

Given: **boolean** withdraw(Account acc, **double** amt) { … }

- Which mechanism should we use?
  - acc.bal >= amt *// if-then-else*
  - amt >= 0 *// if-then-else*
  - acc != null *// if-then-else or @NotNull*
  - Fail to read from or write to database // exception
  - Suppose b is the balance at the entry of withdraw and b' is the balance at the exit of withdraw: b == b' + amt *// assertion*

Part III: File I/O (Self-Study)

The File Class

- 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 filename and its directory path.

```
File myFile = new File("members.txt");
```

Retrieving File Properties

```
var file = new File("image/china.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 Date(file.lastModified()));
```

TestFileClass.java

Text I/O

- A File object encapsulates the properties of a file or a path, but lacks the methods to read/write data from/to a file.
- To read/write data, we create objects using appropriate Java I/O classes, which contain the methods to read/write data from/to a file.

Alternatives:

- Use writeString(Path, String) and readString(Path) of java.nio.Files
- Use the Scanner and PrintWriter I/O classes to read/write strings and numeric values from/to a text file. // old style

Writing Data Using PrintWriter

java.io.PrintWriter	
+PrintWriter(filename: String)	Creates a PrintWriter for the specified file.
+print(s: String): void	Writes a string.
+print(c: char): void	Writes a character.
+print(cArray: char[]): void	Writes an array of character.
+print(i: int): void	Writes an int value.
+print(l: long): void	Writes a long value.
+print(f: float): void	Writes a float value.
+print(d: double): void	Writes a double value.
+print(b: boolean): void	Writes a boolean value.
Also contains the overloaded println methods.	A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §3.6, "Formatting Console Output and Strings."

Major Steps in Writing Data Using PrintWriter

1. Create a File object with a filename

```
var outputFile = new File("COMP3021-teaching-team.txt");
```

2. Use the File object to create a PrintWriter object

```
var writer = new PrintWriter(outputFile);
```

3. Call the PrintWriter object's print/println method to write data

```
writer.print(…);
```

4. Close the PrintWriter object using its close() method

```
writer.close();
```

WriteDataDemo
WriteDataDemoOld.java

Reading Data Using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that produces values scanned from the specified file.
+Scanner(source: String)	Creates a Scanner that produces values scanned from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has another token in its input.
+next(): String	Returns next token as a string.
+nextByte(): byte	Returns next token as a byte.
+nextShort(): short	Returns next token as a short.
+nextInt(): int	Returns next token as an int.
+nextLong(): long	Returns next token as a long.
+nextFloat(): float	Returns next token as a float.
+nextDouble(): double	Returns next token as a double.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern.

Major Steps in Reading Data Using Scanner

1. Create a File object with a filename

```
var inputFile = new File("COMP3021-teaching-team.txt");
```

2. Use the File object to create a Scanner object

```
var sc = new Scanner(inputFile);
```

3. Call the Scanner object's *next* methods to read data

```
members[i] = sc.nextLine();
```

4. Close the Scanner object using its close() method

```
sc.close();
```

ReadFileDemo.java
ReadFileDemoOld.java

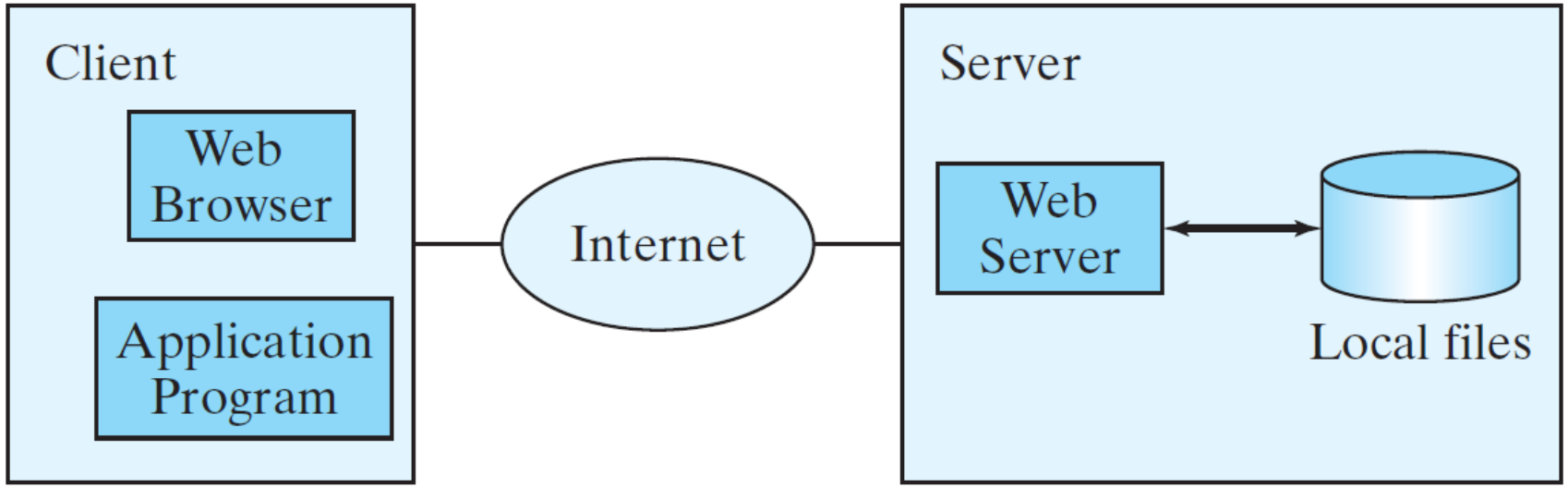
Replacing Text – After Class Exercise

- Write a class named ReplaceText that replaces a string in a text file with a new string. The filename and strings are passed as command-line arguments as follows:
  - java ReplaceText sourceFile targetFile oldString newString
- For example, invoking
  - java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
- replaces all the occurrences of StringBuilder by StringBuffer in FormatString.java and saves the new file in t.txt.

ReplaceText.java

Reading Data from the Web

- Just like you can read data from a file on your computer, you can read data from a file on the Web.



Major Steps in Reading Data from the Web

- Create a URL object with a given http address
  - URL url = **new** URL(“https://www.google.com/index.html");
- Use the URL object’s openStream() method to open an input stream and use the input stream to create a Scanner object
  - **var** input = **new** Scanner(url.openStream());

ReadFileFromURL.java

Supplementary Notes

Review Questions

True or False (From Head First Java)

1. A try block must be followed by a catch and a finally block.
2. If you write a method that might cause a compiler-checked exception, you must wrapped that risky code in a try / catch block.
3. Catch blocks can be polymorphic.
4. Only 'compiler checked' exceptions can be caught.
5. If you define a try / catch block, a matching finally block is optional.
6. If you define a try block, you can pair it with a matching catch or finally block, or both.
7. If you write a method that declares that it can throw a compiler-checked exception, you must also wrap the exception throwing code in a try / catch block.
8. The main() method in your program must handle all unhandled exceptions thrown to it.

1. False, either or both
2. False, you can declare the exception
3. True
4. False, runtime exception can be caught
5. True
6. True, both are acceptable
7. False, the declaration is sufficient
8. False, but if it doesn't, the JVM may shutdown

Review Questions

True or False (From Head First Java)

9. A single try block can have many different catch blocks.
10. A method can only throw one kind of exception.
11. A finally block will run regardless of whether an exception is thrown.
12. A finally block can exist without a try block.
13. A try block can exist by itself, without a catch block or a finally block.
14. Handling an exception is sometimes referred to as 'ducking'.
15. The order of catch blocks never matters.
16. A method with a try block and finally block, can optionally declare the exception.
17. Runtime exceptions must be handled or declared.

9. True
10. False
11. True. It's often used to clean-up partially completed tasks.
12. False
13. False
14. False, ducking is synonymous with declaring
15. False, broadest exceptions must be caught by the last catch blocks
16. False, if you don't have a catch block, you must declare
17. False

Appendix: Why are runtime exceptions unchecked?

- **Question**: Wait just a minute! How come this is the FIRST time we've had to try/catch an Exception? What about the exceptions I've already gotten like NullPointerException and the exception for DivideByZero. I even got a NumberFormatException from the Integer.parseInt() method. How come we didn't have to catch those?

- **Answer**: The compiler cares about all subclasses of Exception, unless they are a special type, RuntimeException. Any exception class that extends RuntimeException gets a free pass. RuntimeExceptions can be thrown anywhere, with or without throws declarations or try/catch blocks. The compiler doesn't bother checking whether a method declares that it throws a RuntimeException, or whether the caller acknowledges that they might get that exception at runtime.

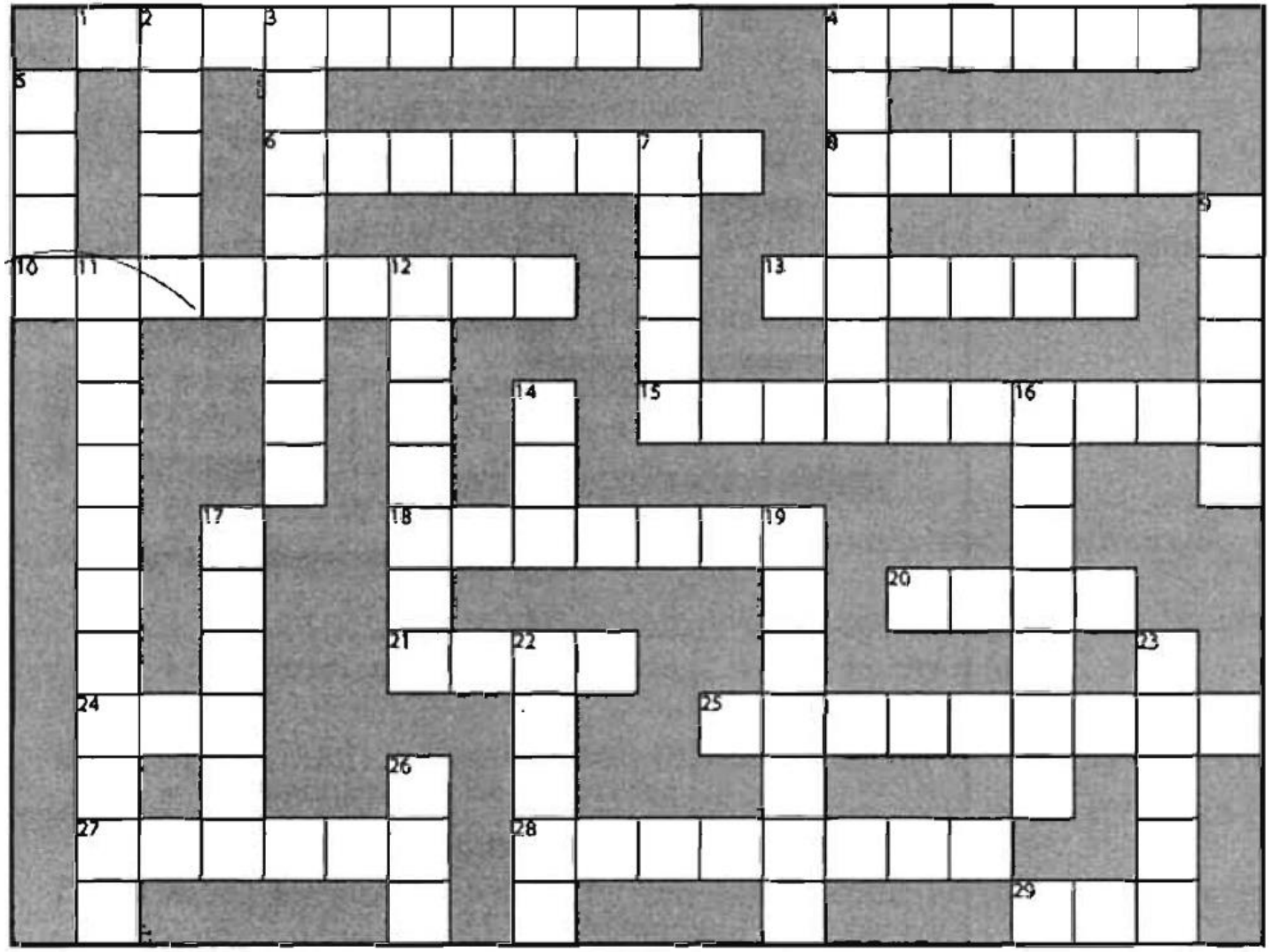
From Head First Java

Appendix: Why are runtime exceptions unchecked?

- **Question**: I'll bite. WHY doesn't the compiler care about those runtime exceptions? Aren't they just as likely to bring the whole show to a stop?
- **Answer**: Most RuntimeExceptions come from a problem in your code logic, rather than a condition that fails at runtime in ways that you cannot predict or prevent. You cannot guarantee the file is there. You cannot guarantee the server is up. But you can make sure your code doesn't index off the end of an array (that's what the .length attribute is for). You WANT RuntimeExceptions to happen at development and testing time. You dont't want to code in a try/catch, for example, and have the overhead that goes with it, to catch something that shouldn't happen in the first place. A try/catch is for handling exceptional situations, not flaws in your code. Use your catch blocks to try to recover from situations you can't guarantee will succeed. Or at the very least, print out a message to the user and a stack trace, so somebody can figure out what happened.

From Head First Java

Java Cross Puzzle



Across

1. To give value
4. Flew off the top
6. All this and more!
8. Start
10. The family tree
13. No ducking
15. Problem objects
18. One of Java's '49'
20. Class hierarchy
21. Too hot to handle
24. Common primitive
25. Code recipe
27. Unruly method action
28. No Picasso here
29. Start a chain of events

Down

2. Currently usable
3. Template's creation
4. Don't show the kids
5. Mostly static API class
7. Not about behavior
9. The template
11. Roll another one off the line
12. Javac saw it coming
14. Attempt risk
16. Automatic acquisition
17. Changing method
19. Announce a duck
22. Deal with it
23. Create bad news
26. One of my roles

extracted from Head First Java

Java Cross Puzzle

extracted from Head First Java

Down

2. Currently usable
3. Template's creation
4. Don't show the kids
5. Mostly static API class
7. Not about behavior
9. The template
11. Roll another one off the line
12. Javac saw it coming
14. Attempt risk
16. Automatic acquisition
17. Changing method
18. One of Java's '49'
19. Announce a duck
22. Deal with it
23. Create bad news
26. One of my roles

Across

1. To give value
4. Flew off the top
6. All this and more!
8. Start
10. The family tree
13. No ducking
15. Problem objects
20. Class hierarchy
21. Too hot to handle
24. Common primitive
25. Code recipe
27. Unruly method action
28. No Picasso here
29. Start a chain of events