

# Exceptions and Text File I/O

Introduction to Programming and  
Computational Problem Solving II

CSE 8B

Lecture 14

# Announcements

- Assignment 6 will be released today
  - Due Mar 5, 11:59 PM

# Exceptions

- Exceptions are runtime errors caused by your program and external circumstances
  - These errors can be caught and handled by your program

# Example: integer divide by zero

```
import java.util.Scanner;

public class Quotient {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        System.out.println(number1 + " / " + number2 + " is " +
            (number1 / number2));
    }
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

# Example: integer divide by zero

- Exception in thread "main" java.lang.ArithmeticException: / by zero
- First approach
  - Mitigate exception with if statement
  - Create a method, so we can reuse it

# Example: integer divide by zero

```
import java.util.Scanner;

public class QuotientWithMethod {
    public static int quotient(int number1, int number2) {
        if (number2 == 0) {
            System.out.println("Divisor cannot be zero");
            System.exit(1);
        }
        return number1 / number2;
    }

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        int result = quotient(number1, number2);
        System.out.println(number1 + " / " + number2 + " is "
            + result);
    }
}
```

Problem:  
a method should  
never terminate  
a program

# Example: integer divide by zero

- Exception in thread "main" java.lang.ArithmeticException: / by zero
- First approach
  - Mitigate exception with if statement
  - Create a method, so we can reuse it
    - Problem: a method should never terminate a program
- Second approach
  - Have the method notify the caller

# Example: integer divide by zero

```
import java.util.Scanner;

public class QuotientWithException {
    public static int quotient(int number1, int number2) {
        if (number2 == 0)
            throw new ArithmeticException("Divisor cannot be zero");

        return number1 / number2;
    }

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        // Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        try {
            int result = quotient(number1, number2);
            System.out.println(number1 + " / " + number2 + " is "
                + result);
        }
        catch (ArithmeticException ex) {
            System.out.println("Exception: an integer " +
                "cannot be divided by zero ");
        }

        System.out.println("Execution continues ...");
    }
}
```

Throw an  
ArithmeticException

Try something that may  
throw an exception

Catch an  
ArithmeticException

Handle the caught  
exception in the catch block



# Exception handling

- Exception handling enables a method to throw an exception to its caller
- Without this capability, a method must handle the exception or terminate the program
- Separates
  - The detection of an error
  - The handling of an error

# Exception types

- Exceptions are objects
  - Remember, objects are instances of classes

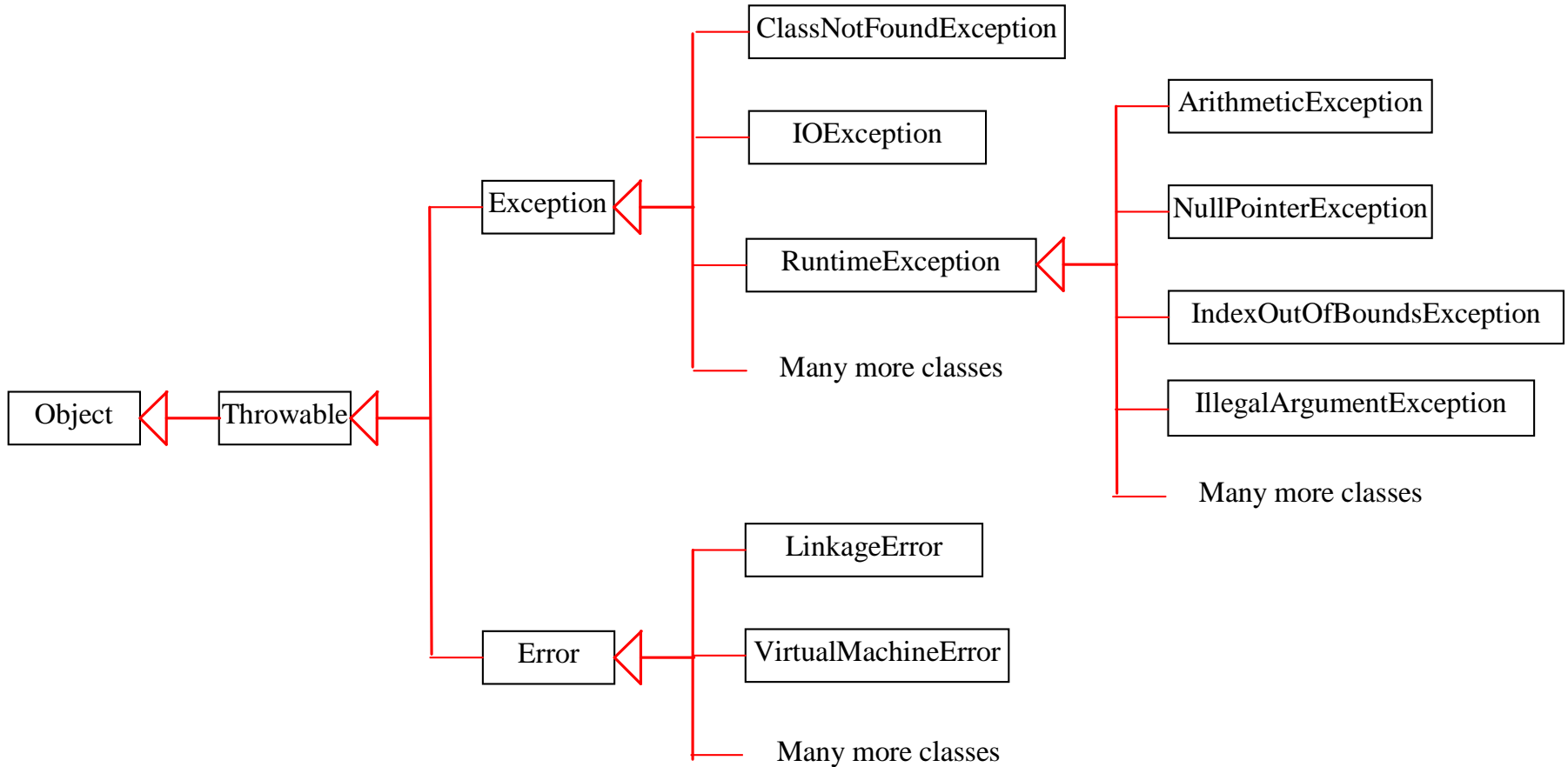
- The root class for exception is `java.lang.Throwable`

<https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Throwable.html>

- Three major types
  - System errors
  - Exceptions
  - Runtime Exceptions

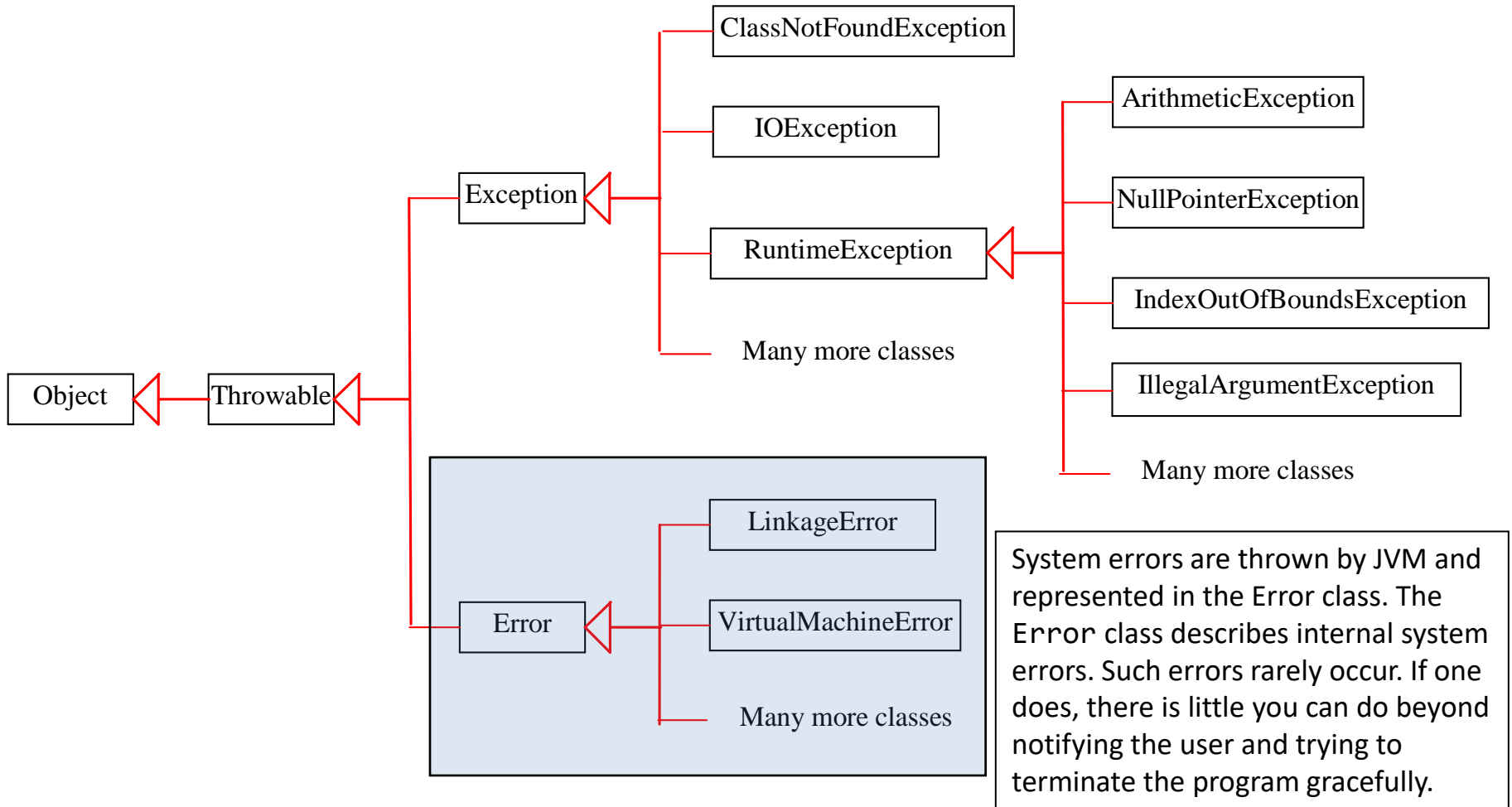
# Exception types



# Error

<https://docs.oracle.com/javase/8/docs/api/java/lang/Error.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Error.html>

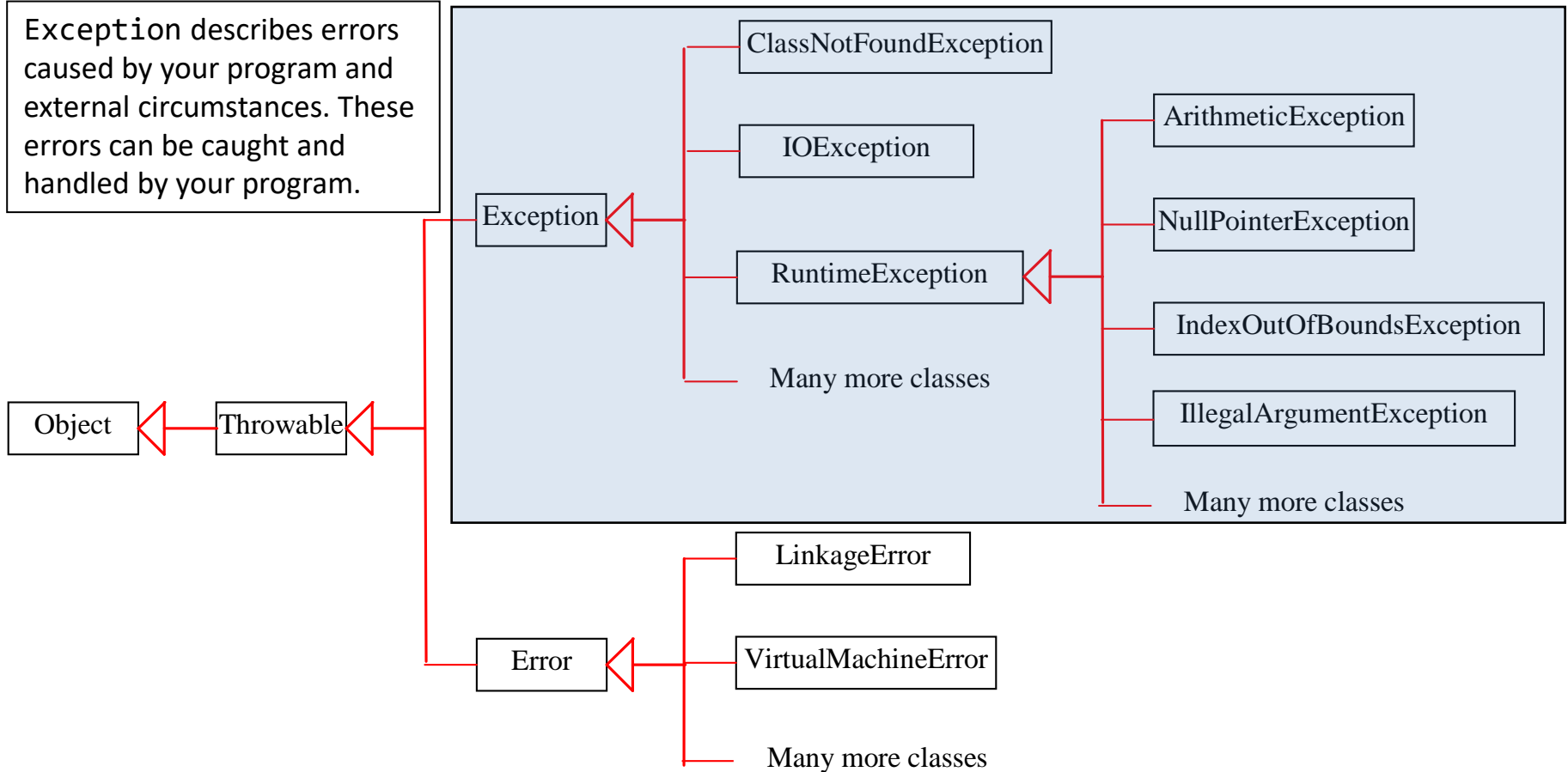


# Exception

<https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Exception.html>

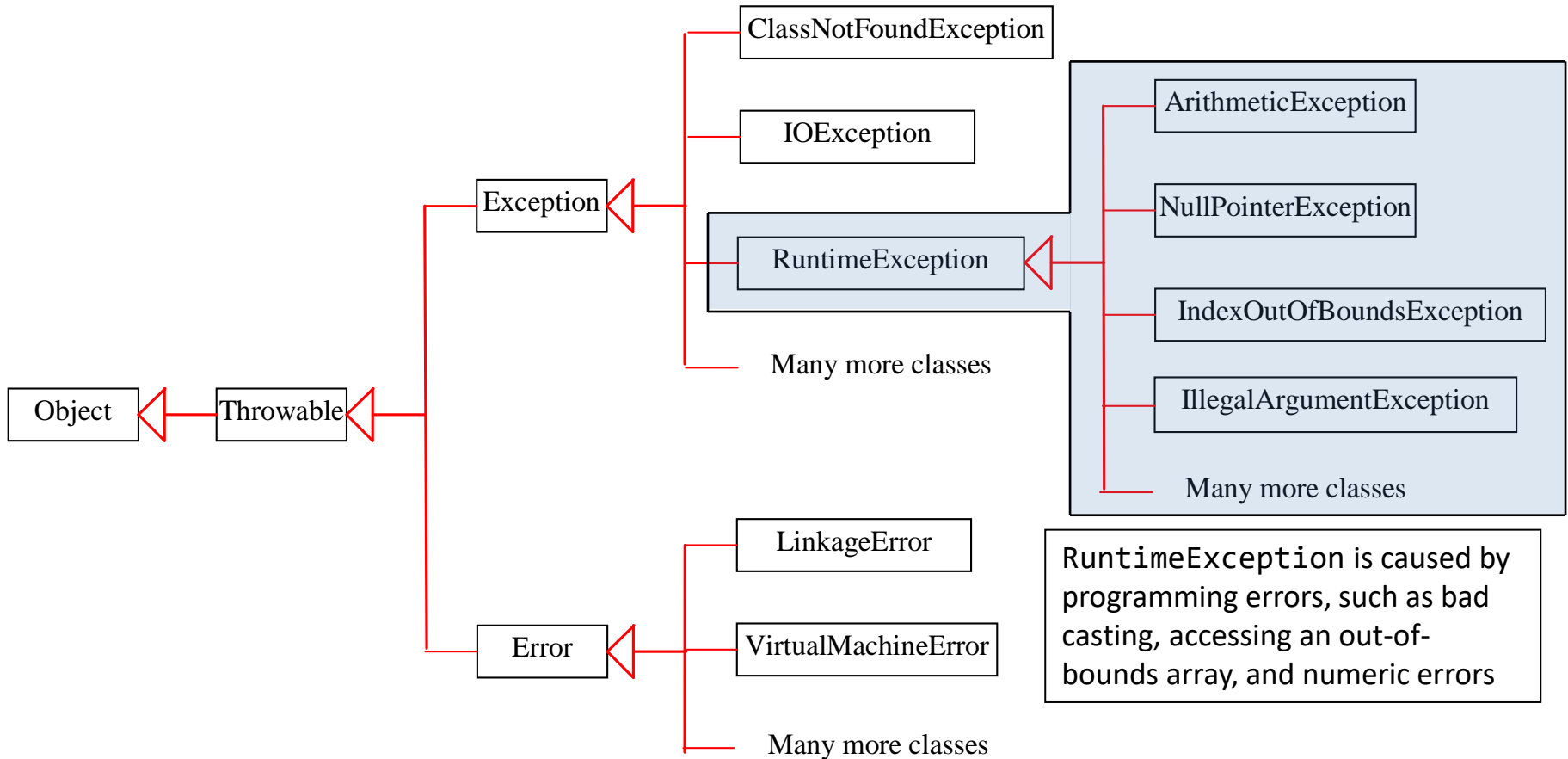
Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



# RuntimeException

<https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/RuntimeException.html>



# Exception types

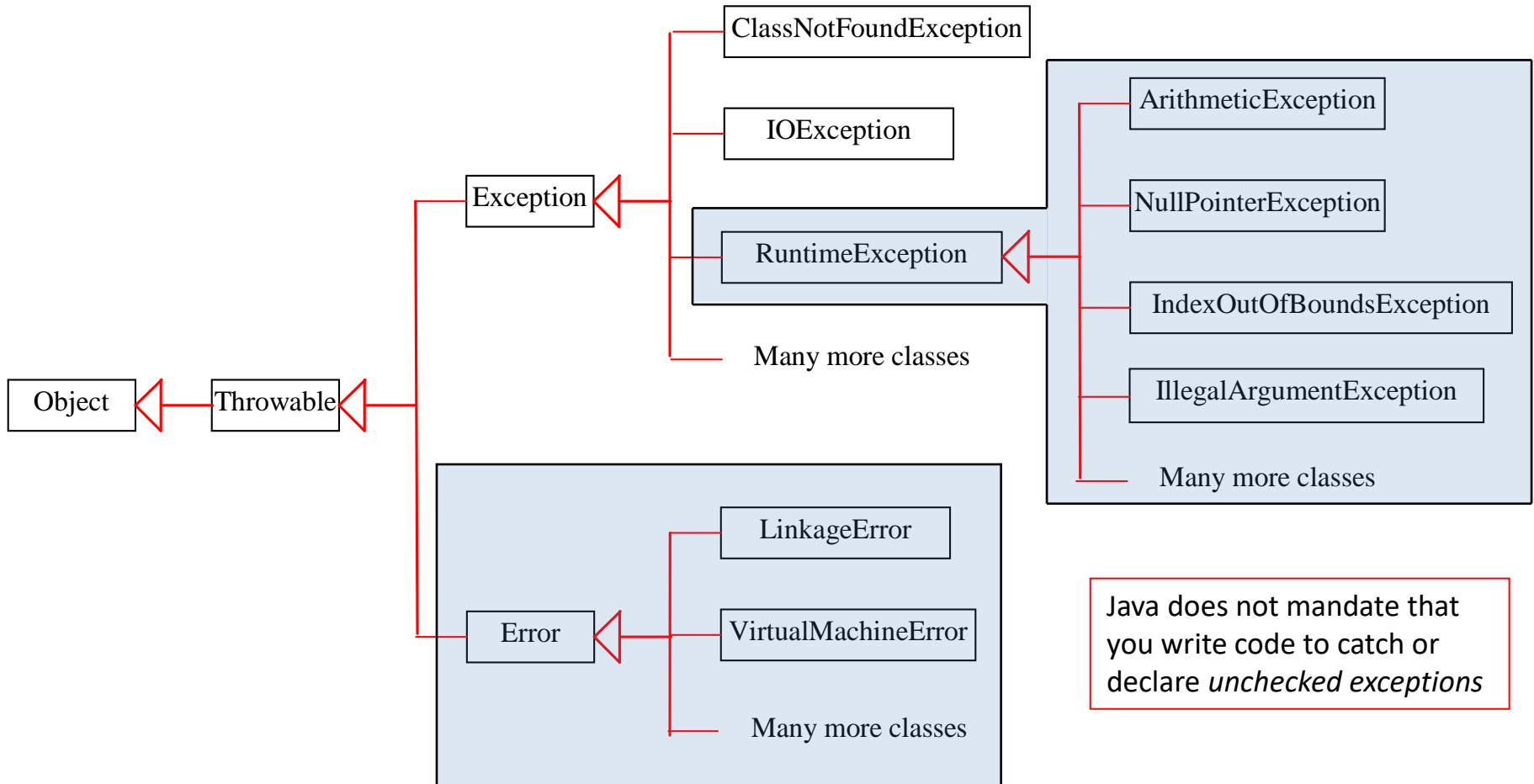
- Exceptions are objects
  - Remember, objects are instances of classes
- The root class for exception is `java.lang.Throwable`
  - All Java exception classes inherit directly or indirectly from `Throwable`
  - Use overridden `getMessage()` member method
- You can create your own exception classes by extending `Exception` or a subclass of `Exception`

# Unchecked exceptions vs. checked exceptions

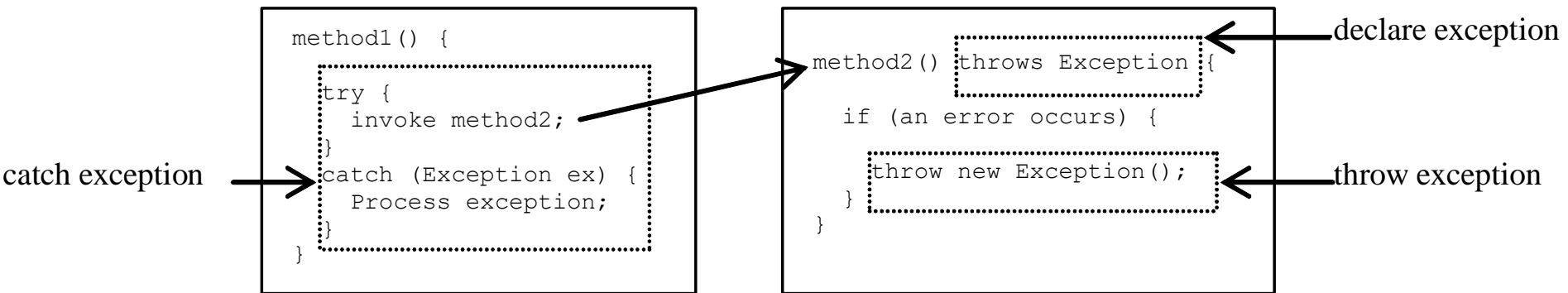
- `RuntimeException`, `Error`, and their subclasses are known as *unchecked exceptions*
  - Usually programming logic errors that are unrecoverable
  - These should be corrected in the program
- All other exceptions are known as *checked exceptions*
  - **The compiler forces the programmer to check and deal with these exceptions**



# Unchecked exceptions



# Declaring, throwing, and catching exceptions



# Declaring exceptions

- Every method must state the types of *checked exceptions* it might throw
  - This is called *declaring exceptions*

- Examples

```
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 called *throwing an exception*

- For example

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

# Catching exceptions

- When an exception is thrown, it can be caught and handled in a `try-catch` block
  - If no exceptions are thrown in the `try` block, then the catch blocks are skipped
- If an exception is thrown in the `try` block, Java **skips the remaining statements in the `try` block** and starts the process of finding the code to handle the exception
  - This is called *catching an exception*

# Catching exceptions

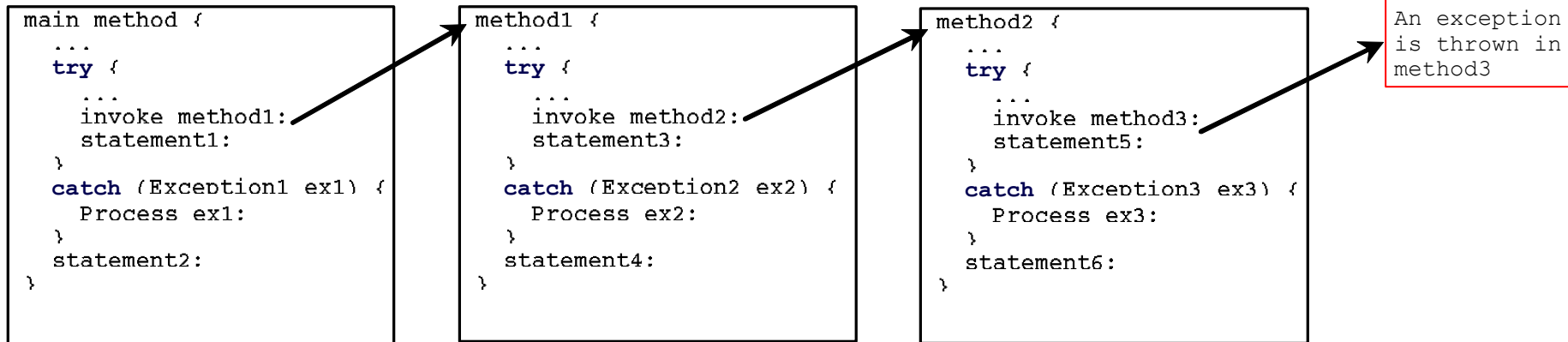
```
try {  
    // Statements that may throw exceptions  
}  
catch (Exception1 exVar1) {  
    // Handler for Exception1  
}  
catch (Exception2 | Exception3 | ... | ExceptionK exVar)  
{  
    // Same code for handling these exceptions  
}  
...  
catch (ExceptionN exVarN) {  
    // Handler for ExceptionN  
}
```

**The order exceptions are specified is important.** A compile error occurs if a catch block for a superclass type appears before a catch block for a subclass type.

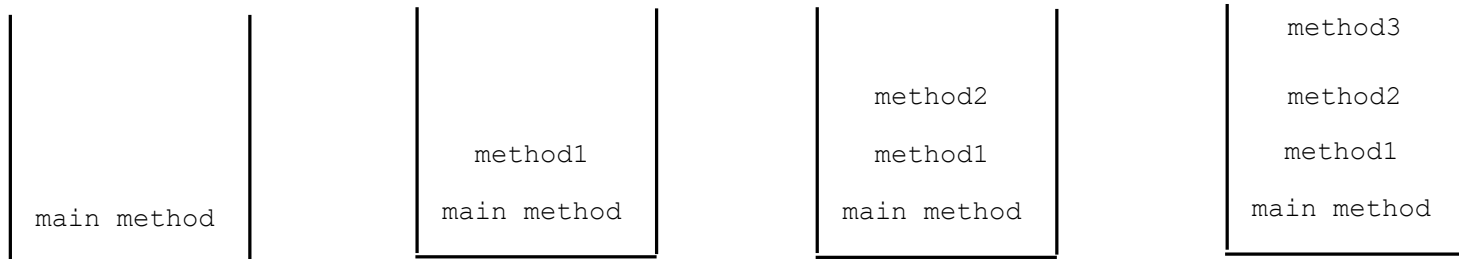
If no handler is found, then the program terminates and prints an error message on the console

# Catching exceptions

- The code handling the exception is called the *exception handler*
  - It is found by *propagating the exception* backward through the call stacks, starting from the current method



Call Stack



# Checked exceptions

- Remember, the compiler forces the programmer to check and deal with checked exceptions (i.e., any exception other than `Error` or `RuntimeException`)
- If a method declares a checked exception, you must invoke it in a `try-catch` block or declare to throw the exception in the calling method

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

(a)

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

(b)



# The `finally` clause

- The `finally` clause is always executed, regardless of whether an exception occurred

```
try {  
    // statements  
}  
catch(TheException ex) {  
    // handling statements  
}  
finally {  
    // final statements  
}
```

# 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 {  
    // statements  
}  
catch(TheException ex) {  
    // handling statements before rethrowing  
    throw ex;  
}
```

- You can also throw a new exception along with the original exception
  - This is called *chained exceptions*
    - <https://docs.oracle.com/javase/tutorial/essential/exceptions/chained.html>

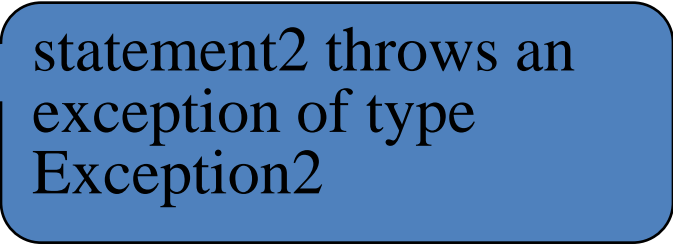
# Trace code

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

nextStatement;
```

# Trace code

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



statement2 throws an  
exception of type  
Exception2

# Trace code

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



Handling exception

# Trace code

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

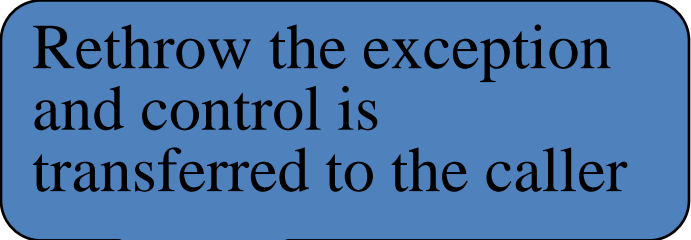


Execute the final block

nextStatement;

# Trace code

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



Rethrow the exception  
and control is  
transferred to the caller

# When to use a try-catch block

- Use a try-catch block to deal with **unexpected** error conditions
- Do not use it to deal with simple, **expected** situations
  - For example, use this

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

instead of this

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



# When to throw exceptions

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

# Defining custom exception classes

- Use the exception classes in the Java API whenever possible
- If the predefined classes are insufficient, then you can define a custom exception class by extending the `java.lang.Exception` class

# Exception handling

- Exception handling separates error-handling code from normal programming tasks
  - Makes programs easier to read and to modify
- The **try** block contains the code that is executed in **normal** circumstances
- The **catch** block contains the code that is executed in **exceptional** circumstances
- A method should **throw** an exception if the error needs to be handled by its caller
- Warning: 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

# Text I/O

- In order to perform I/O, you need to create objects using appropriate Java I/O classes
  - The objects contain the methods for reading/writing data from/to a file

## File

- <https://docs.oracle.com/javase/8/docs/api/java/io/File.html>
- <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/File.html>

## Scanner

- <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>
- <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Scanner.html>

## PrintWriter

- <https://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html>
- <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/PrintWriter.html>

# Absolute file names

- Absolute file name includes full path

- Unix

- `/home/bochoa/cse/assignment7/Assignment7.java`

- Java string

- `String pathname =  
"/home/bochoa/cse/assignment7/Assignment7.java";`

- Windows

- `C:\cse\assignment7\Assignment7.java`

- Java string

- `String pathname =  
"C:\\cse\\assignment7\\Assignment7.java";`

# Relative file names

- Relative file name includes path relative to working directory

- For example, if you are in directory cse

- Unix

- assignment7/Assignment7.java

- Java string

- ```
String pathname = "assignment7/Assignment7.java"
```

- Windows

- assignment7\Assignment7.java

- Java string

- ```
String pathname = "assignment7\\Assignment7.java"
```

# The `File` class

- The `File` class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files (and directories) in a machine-independent fashion
- The path name is a string
- The `File` class is a wrapper class for the file (or directory)

java.io.File
+File(pathname: String)
+File(parent: String, child: String)
+File(parent: File, child: String)
+exists(): boolean
+canRead(): boolean
+canWrite(): boolean
+isDirectory(): boolean
+isFile(): boolean
+isAbsolute(): boolean
+isHidden(): boolean
+getAbsolutePath(): String
+getCanonicalPath(): String
+getName(): String
+getPath(): String
+getParent(): String
+lastModified(): long
+length(): long
+listFile(): File[]
+delete(): boolean
+renameTo(dest: File): boolean
+mkdir(): boolean
+mkdirs(): boolean

Creates a File object for the specified path name. The path name may be a directory or a file.
Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.
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.
Returns true if the file or the directory represented by the File object exists.
Returns true if the file represented by the File object exists and can be read.
Returns true if the file represented by the File object exists and can be written.
Returns true if the File object represents a directory.
Returns true if the File object represents a file.
Returns true if the File object is created using an absolute path name.
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.
Returns the complete absolute file or directory name represented by the File object.
Returns the same as <code>getAbsolutePath()</code> except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
Returns the last name of the complete directory and file name represented by the File object. For example, <code>new File("c:\\book\\test.dat").getName()</code> returns <code>test.dat</code> .
Returns the complete directory and file name represented by the File object. For example, <code>new File("c:\\book\\test.dat").getPath()</code> returns <code>c:\\book\\test.dat</code> .
Returns the complete parent directory of the current directory or the file represented by the File object. For example, <code>new File("c:\\book\\test.dat").getParent()</code> returns <code>c:\\book</code> .
Returns the time that the file was last modified.
Returns the size of the file, or 0 if it does not exist or if it is a directory.
Returns the files under the directory for a directory File object.
Deletes the file or directory represented by this File object. The method returns true if the deletion succeeds.
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.
Creates a directory represented in this File object. Returns true if the the directory is created successfully.
Same as <code>mkdir()</code> except that it creates directory along with its parent directories if the parent directories do not exist.



# The File class example

```
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()));
    }
}
```

# File text I/O

- A `File` object encapsulates the properties of a file (or directory), *but does not contain the methods for reading/writing data from/to a file*
- In order to perform I/O, you need to create objects using appropriate Java I/O classes
  - The objects contain the methods for reading/writing data from/to a file
- Use the `Scanner` class for reading text data from a file
- Use the `PrintWriter` class for writing text data to a file

# Reading data from the console

- Create a Scanner object

```
Scanner input = new Scanner(System.in);
```

- Example

```
System.out.print("Enter a double value: ");  
Scanner input = new Scanner(System.in);  
double d = input.nextDouble();
```

# Reading data using Scanner

- Reading data from the console

```
Scanner input = new Scanner(System.in);
```

- Reading data from a file

```
Scanner input = new Scanner(new File(filename));
```

# Reading data using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner object to read data from the specified file.
+Scanner(source: String)	Creates a Scanner object to read data 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.

# Reading data from a file

```
public class ReadData {
    public static void main(String[] args) throws Exception {
        // Create a File instance
        java.io.File file = new java.io.File("scores.txt");

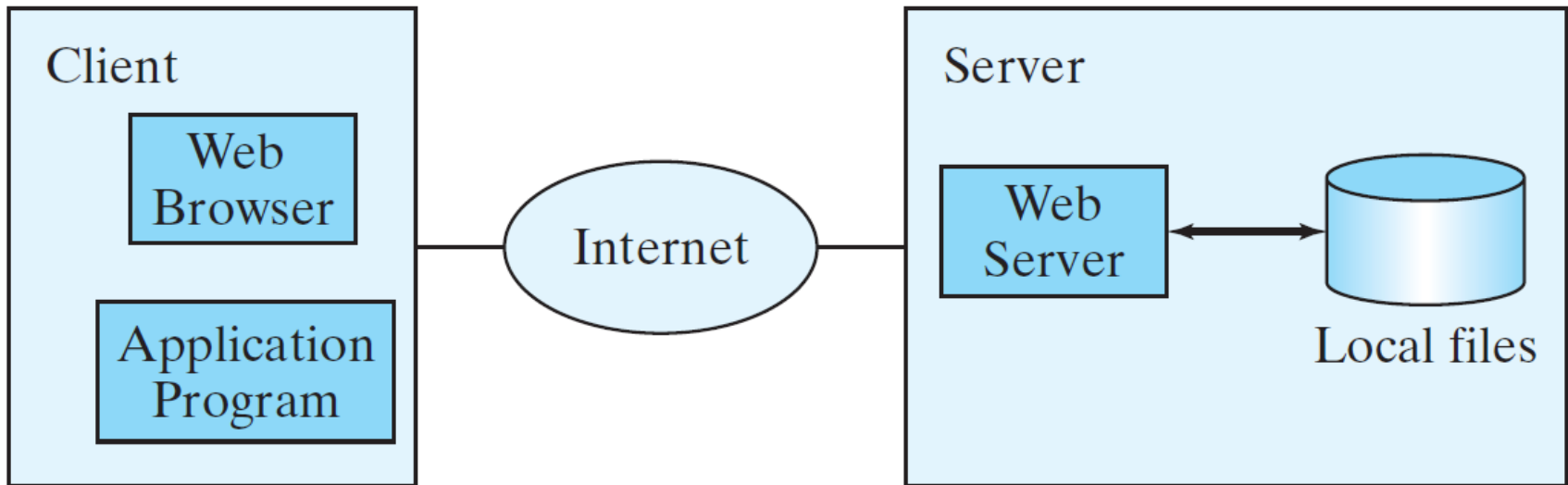
        // Create a Scanner for the file
        Scanner input = new Scanner(file);

        // Read data from a file
        while (input.hasNext()) {
            String firstName = input.next();
            String mi = input.next();
            String lastName = input.next();
            int score = input.nextInt();
            System.out.println(
                firstName + " " + mi + " " + lastName + " " + score);
        }

        // Close the file
        input.close();
    }
}
```

# Reading data from the internet

- Just like you can read data from a file on the computer, you can read data from a file on the internet



# Reading data from the internet

```
public class ReadFileFromURL {
    public static void main(String[] args) {
        System.out.print("Enter a URL: ");
        String urlString = new Scanner(System.in).next();

        try {
            java.net.URL url = new java.net.URL(urlString);
            int count = 0;
            Scanner input = new Scanner(url.openStream());
            while (input.hasNext()) {
                String line = input.nextLine();
                count += line.length();
            }

            System.out.println("The file size is " + count + " characters");
        }
        catch (java.net.MalformedURLException ex) {
            System.out.println("Invalid URL");
        }
        catch (java.io.IOException ex) {
            System.out.println("IO Errors");
        }
    }
}
```



# 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.

See lecture 4

}

# Writing data to a file

```
public class WriteData {
    public static void main(String[] args) throws java.io.IOException {
        java.io.File file = new java.io.File("scores.txt");
        if (file.exists()) {
            System.out.println("File already exists");
            System.exit(0);
        }

        // Create a file
        java.io.PrintWriter output = new java.io.PrintWriter(file);

        // Write formatted output to the file
        output.print("John T Smith ");
        output.println(90);
        output.print("Eric K Jones ");
        output.println(85);

        // Close the file
        output.close();
    }
}
```

# Use try-with-resources syntax

- When reading or writing programmers often forget to close the file
- The try-with-resources syntax automatically closes the files

## – Write file example

```
try (  
    // Create a file  
    java.io.PrintWriter output = new java.io.PrintWriter(file);  
) {  
    // Write formatted output to the file  
    output.print("John T Smith ");  
    output.println(90);  
    output.print("Eric K Jones ");  
    output.println(85);  
}
```

Semicolon is optional  
if a single statement



# File I/O example

```
public class ReplaceText {
    public static void main(String[] args) throws Exception {
        // Check command line parameter usage
        if (args.length != 4) {
            System.out.println(
                "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
            System.exit(1);
        }

        // Check if source file exists
        File sourceFile = new File(args[0]);
        if (!sourceFile.exists()) {
            System.out.println("Source file " + args[0] + " does not exist");
            System.exit(2);
        }

        // Check if target file exists
        File targetFile = new File(args[1]);
        if (targetFile.exists()) {
            System.out.println("Target file " + args[1] + " already exists");
            System.exit(3);
        }

        try (
            // Create input and output files
            Scanner input = new Scanner(sourceFile);
            PrintWriter output = new PrintWriter(targetFile);
        ) {
            while (input.hasNext()) {
                String s1 = input.nextLine();
                String s2 = s1.replaceAll(args[2], args[3]);
                output.println(s2);
            }
        }
    }
}
```

# Next Lecture

- Abstract classes