Lecture 11

Exception Handling and Text IO

- When a program runs into a runtime error, the program terminates abnormally.

- How can you handle the runtime error so that the program can continue to run or terminate gracefully?

```
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(); //1;
    int number2 = input.nextInt(); //0;

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

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

      // Prompt the user to enter two integers
```

Exception in thread "main"
java.lang.ArithmeticException: / by zero
      at Quotient.main(Quotient.java:12)

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

```java
public class QuotientWithIf {
  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(); //1;
    int number2 = input.nextInt(); //0;
    if (number2 != 0)
      System.out.println(number1 + " / " + number2 + " is " +
        (number1 / number2));
    else
      System.out.println("Divisor cannot be zero ");
  }
}
```

```java
public class QuotientWithException {
  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(); //1
    int number2 = input.nextInt(); //0

    try {
      int result = 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 ...");
  }
}
```

- It enables a method to throw an exception to its caller.

- Without this capability, a method must handle the exception or terminate the program.

NYU

```java
public class InputMismatchExceptionDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean continueInput = true;

    do {
      try {
        System.out.print("Enter an integer: ");
        int number = input.nextInt();

        // Display the result
        System.out.println(
          "The number entered is " + number);

        continueInput = false;
      }
      catch (InputMismatchException ex) {
        System.out.println("Try again. (" +
          "Incorrect input: an integer is required)");
        input.nextLine(); // discard input
      }
    } while (continueInput);
  }
}
```
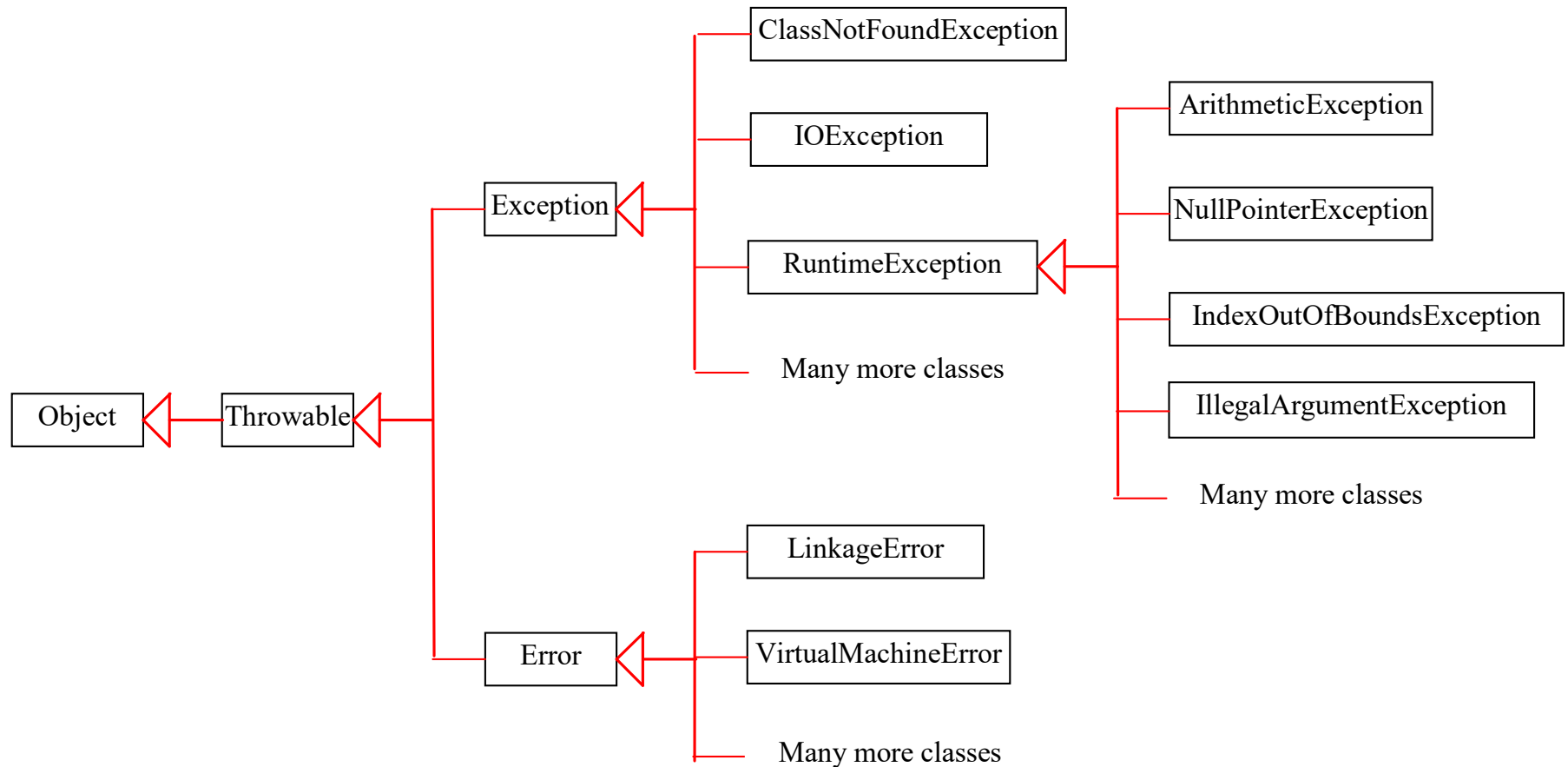
NYU

- By handling `InputMismatchException`, your program will continuously read an input until it is correct.

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

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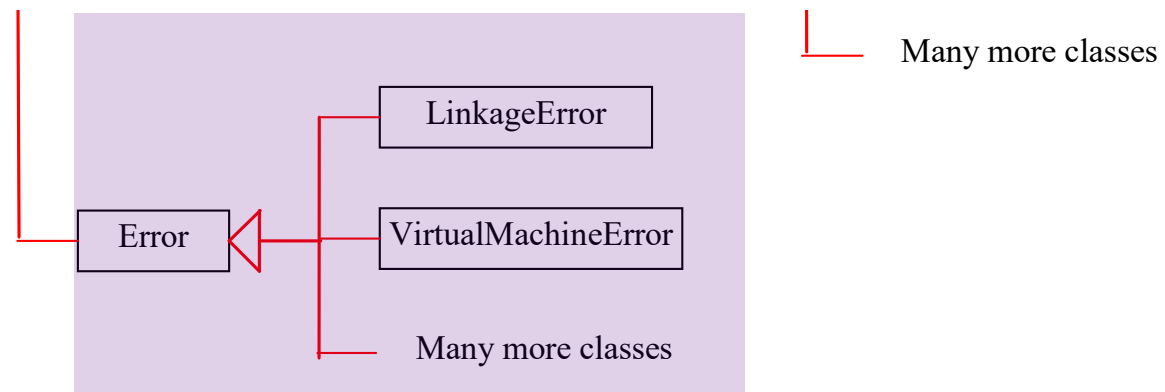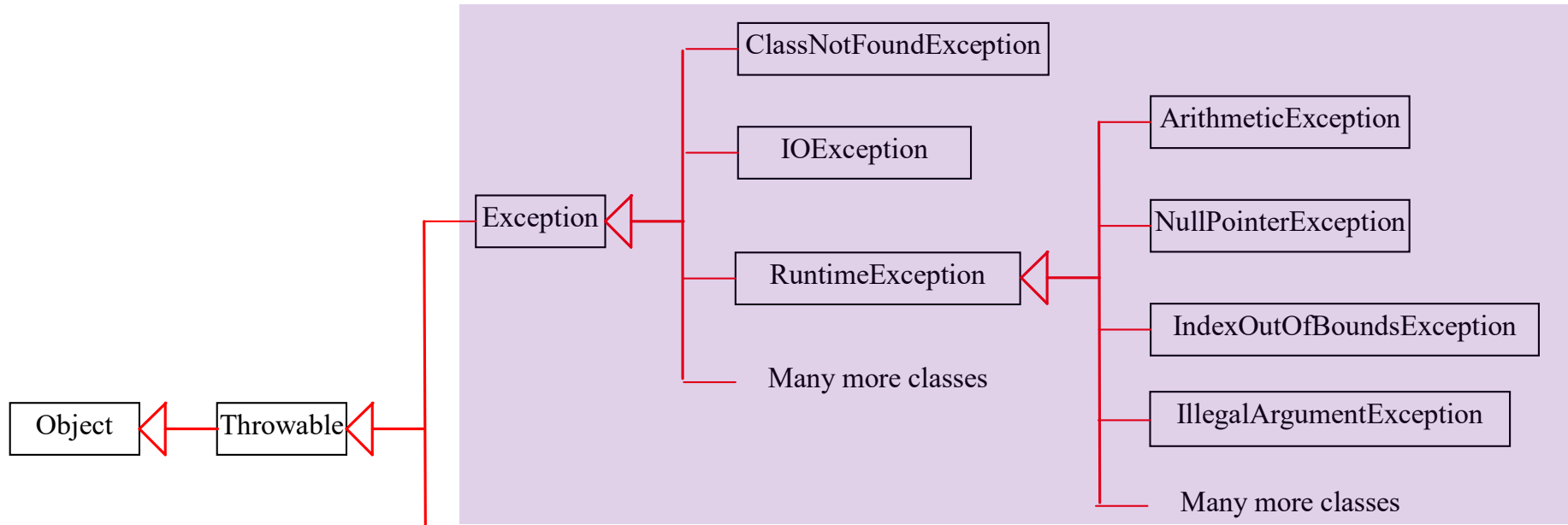
- `Exception` describes errors caused by your program and external circumstances.
- These errors can be caught and handled by your program.

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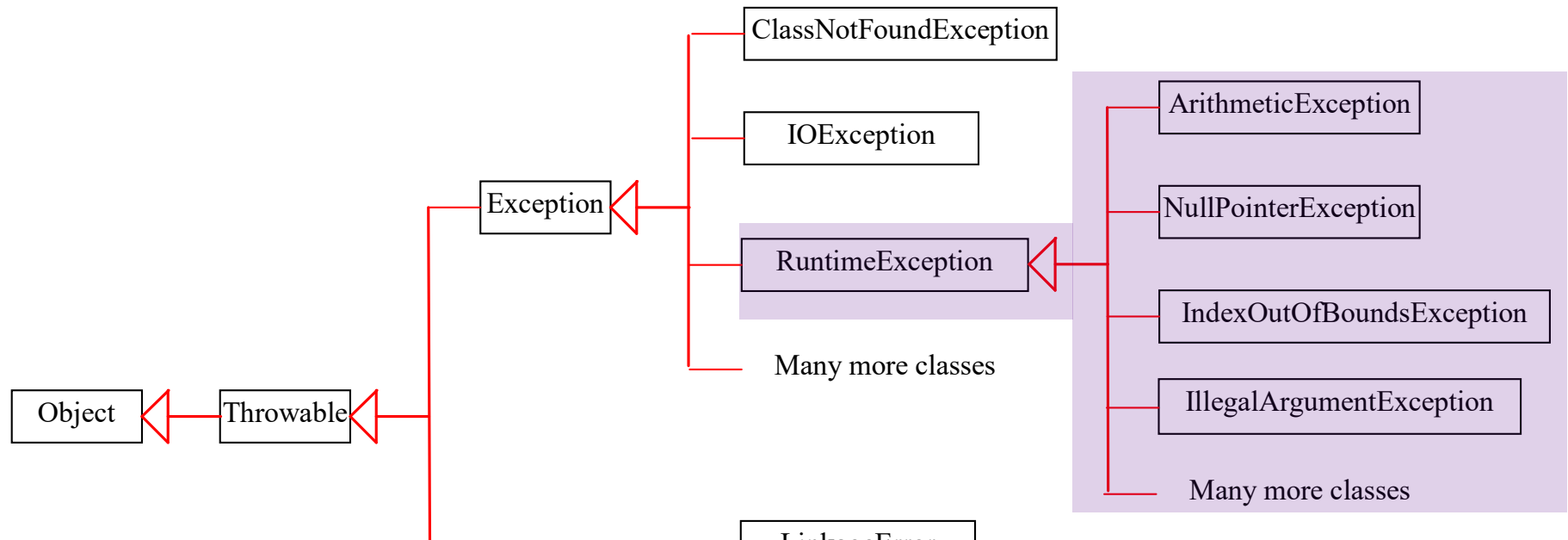
- `RuntimeException` is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

- `RuntimeException, Error` and their subclasses are known as **unchecked exceptions**.
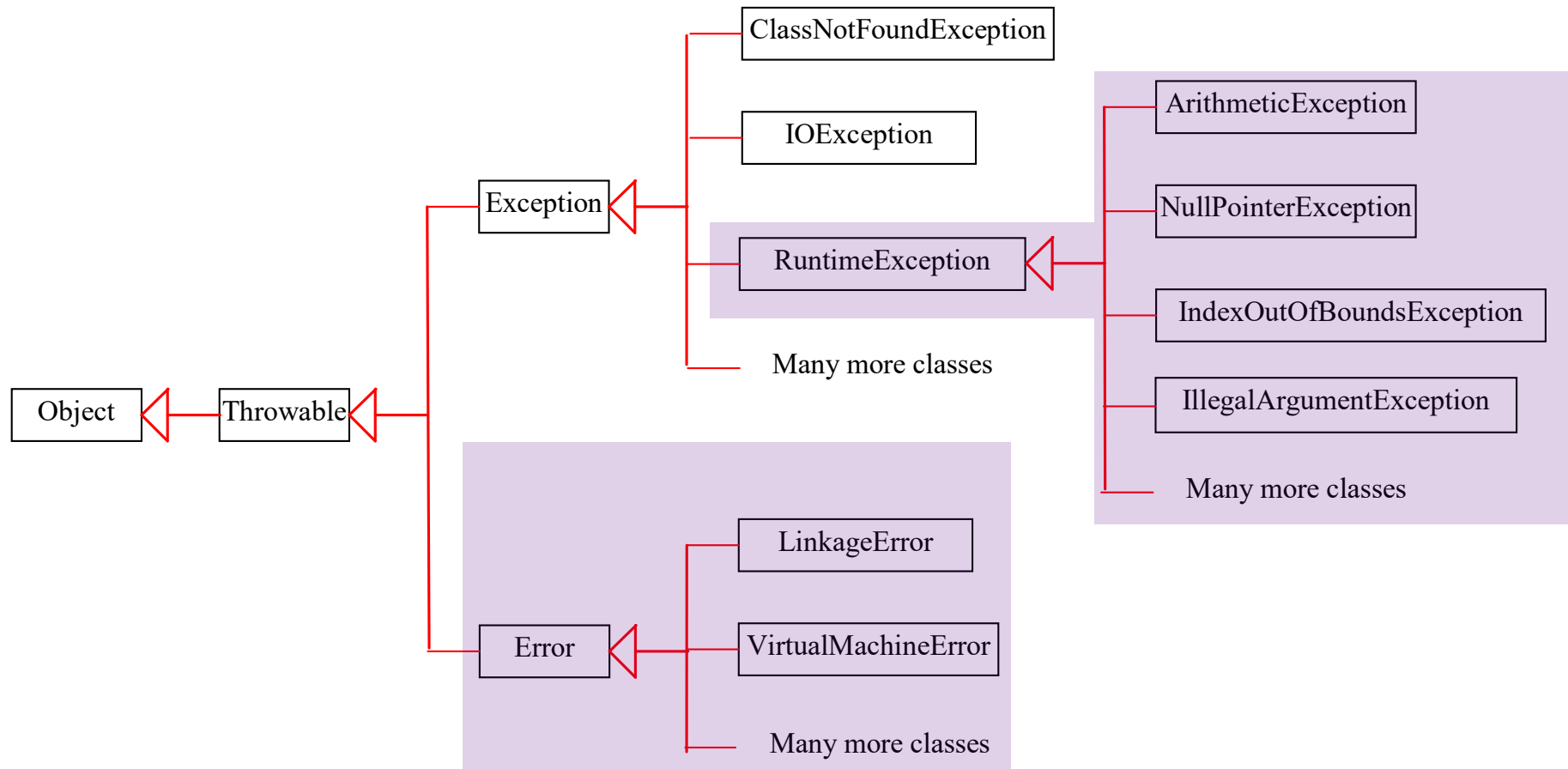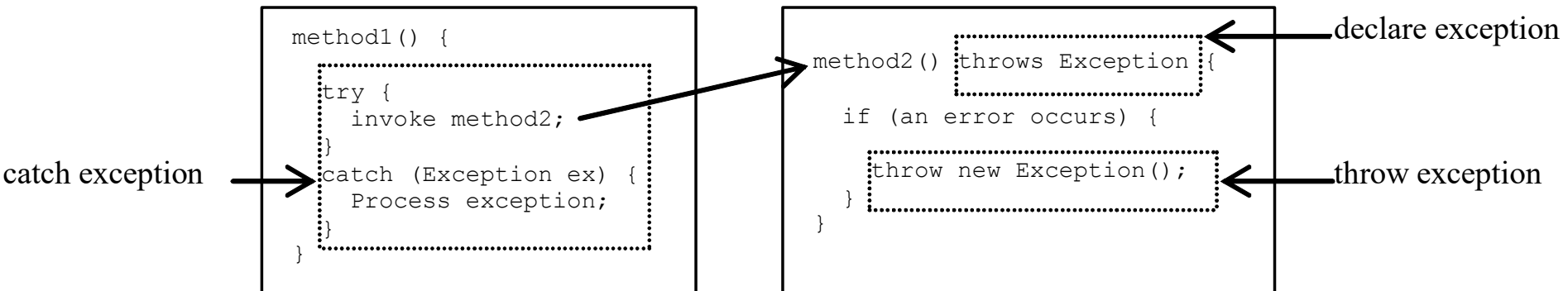
- All other exceptions are known as checked exceptions, meaning that the compiler forces the programmer to check and deal with the exceptions.

- 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.
- 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.

NYU

ClassNotFoundException

IOException

Exception

RuntimeException

Many more classes

Object

Throwable

ArithmeticException

NullPointerException

IndexOutOfBoundsException

IllegalArgumentException

Many more classes

Error

LinkageError

VirtualMachineError

Many more classes

```
method1() {

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

catch exception

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

declare exception

throw exception

- Every method must state the types of checked exceptions it might throw.

- This is known as declaring exceptions.

- Example:

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

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

- When the program detects an error, the program can create an instance of an appropriate exception type and throw it.

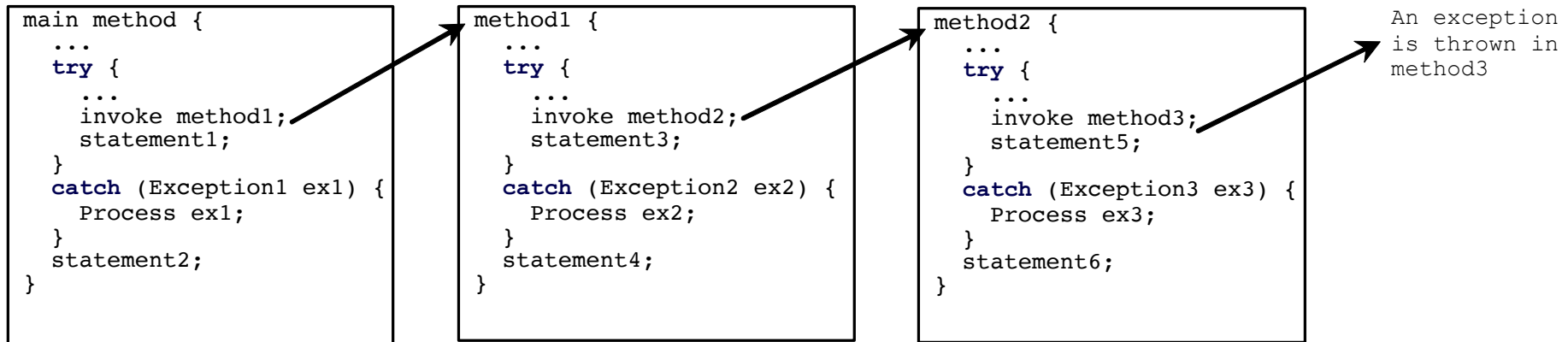- This is known as throwing an exception.

- Example:
  ```
  throw new TheException();
  ```

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

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

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

```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

An exception
is thrown in
method3

Call Stack

```
|              |
|              |
|              |
| main method  |
```

```
|              |
|              |
| method1      |
| main method  |
```

```
|              |
| method2      |
| method1      |
| main method  |
```

```
| method3      |
| method2      |
| method1      |
| main method  |
```

- Java forces you to deal with checked exceptions.

- If a method declares a checked exception (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` that may throw a checked exception (e.g., `IOException`), you have 2 possibilities

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

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

```java
public class Circle {
  private static int numberOfObjects = 0;

  public Circle(double newRadius) {
    setRadius(newRadius);
    numberOfObjects++;
  }

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

```
public class TestCircle {
  public static void main(String[] args) {
    try {
      Circle c1 = new Circle(5);
      Circle c2 = new Circle(-5);
      Circle c3 = new Circle(0);
    }
    catch (IllegalArgumentException ex) {
      System.out.println(ex);
    }

    System.out.println("created: " +
      Circle.getNumberOfObjects());
  }
}
```

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

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

NYU

Suppose no exceptions in the statements

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

nextStatement;
```

NYU

```
try {

    statements;

}

catch(TheException ex) {

    handling ex;

}

finally {

    finalStatements;

}


nextStatement;
```

The final block is always executed

NYU

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

Next statement in the method is executed

```
nextStatement;
```

Suppose an exception of type `Exception1` is thrown in `statement2`

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

nextStatement;
```

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

nextStatement;
```

The exception is handled.

NYU

The final block is always executed.

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

nextStatement;
```

NYU

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

nextStatement;
```

The next statement in the method is now executed.

statement2 throws an exception of type Exception2.

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

nextStatement;
```

NYU

Handling exception

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

nextStatement;
```

Execute the final block

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

nextStatement;
```

NYU

Rethrow the exception and control is transferred to the caller

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

nextStatement;
```

- 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.

NYU

- 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 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.

NYU

- **Bad**

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

- **Better**

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

- 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`.

- 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.

- An assertion is declared using the new Java keyword assert in JDK 1.4
  - `assert assertion; or`
  - `assert assertion : detailMessage;`

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

- 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.

- `assert assertion;` (no detail message)
  - the no-arg constructor of `AssertionError` is used.

- `assert assertion : detailMessage;`
  - 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.

```
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;
  }
}
```

- 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.

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

- 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

- 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.

- 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 in the Circle class should be rewritten using exception handling.

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

- 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.

- Another good use of assertions is place assertions in a switch statement without a default case.

- Example:

```
switch (month) {
   case 1: ... ; break;
   case 2: ... ; break;
   ...
   case 12: ... ; break;
   default: assert false : "Invalid month: " +
month
}
```

- 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 path name. The path name 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 file name 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 *hidden* 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 path name, 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.
+listFile(): 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 the directory is created successfully.
+mkdirs(): boolean	Same as mkdir() except that it creates directory along with its parent directories if the parent directories do not exist.

```
java.io.File file = ...

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()));
```

- A File object encapsulates the properties of a file or a path, 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.

- How to read/write strings and numeric values from/to a text file using the `Scanner` and `PrintWriter` classes.

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 §4.6, "Formatting Console Output and Strings."

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

- Programmers often forget to close the file.

- JDK 7 provides the followings new try-with-resources syntax that automatically closes the files.

```
try (declare and create resources) {
   Use the resource to process the file;
}
```

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

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

| 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.

```java
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();
      int score = input.nextInt();
      System.out.println(firstName + " " + score);
    }

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

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

NYU
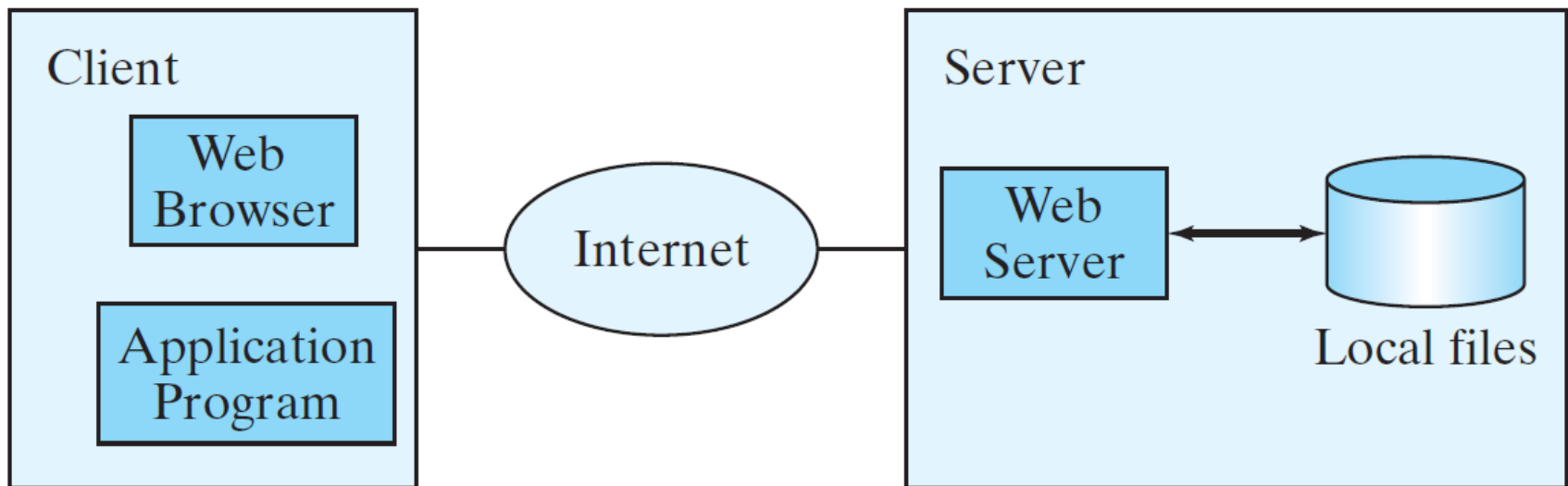
- `URL url = new URL("www.google.com/index.html");`

- After a URL object is created, you can use the `openStream()` method defined in the URL class to open an input stream

- Use this stream to create a Scanner object as follows:
  - `Scanner input = new Scanner(url.openStream());`

```java
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();
      }
      input.close();
      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");
    }
  }
}
```