CIS2571 – Intro to Java

Chapter14 → Exception Handling and Text I/O

Information consolidated from a variety of textbook and online resources

Topic Objectives

- Runtime Errors and Exception Handling
- Exception Handling in Java
- Exception Types
- Java Exception Handling Model
- Getting Information from Exceptions
- The finally Clause
- Rethrowing Exceptions
- Chained Exceptions
- Creating Custom Exception Classes
- File Class
- PrintWriter Class
- Scanner Class
- JFileChooser Class
- Reading Data from the Web

Runtime Errors and Exception Handling

- Runtime errors occur while program is executing
  - Runtime environment detects an illegal operation
  - Program terminates abnormally
- Runtime errors caused by exceptions
  - Exception is an object that represents an error or condition that prevents normal execution from occurring
  - If exception is not handled, program terminates abnormally
- Advantages of exception handling
  - Enables method to throw exception to calling method
  - Calling method has option to:
    - handle exception, **or**
    - terminate program

Runtime Errors and Exception Handling

- Separate detection of error from handling of error
  - Separate error-handling code from application specific logic
- When to use exceptions:
  - If you want exception to be **processed by calling method**, exception should be **created** and **thrown**
  - If you **handle the exception** in the method where it occurs, there is no need to throw or use exceptions
- Use a try-catch block when you have to deal with unexpected error conditions that could generate an exception

→Examples

Example With No Exception Handling

```
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));
   }
}
```

*Note that floating-point number divided by 0 does **not** raise exception. Result is the special value NaN (Not a Number), NEGATIVE_INFINITY, or POSITIVE_INFINITY (defined in Float and Double wrapper classes).*

```
Enter two integers: 5 2 <Enter>
5 / 2 is 2
```

```
Enter two integers: 3 0 <Enter>
Exception in thread "main"
java.lang.ArithmeticException: / by
zero at Quotient.mainQuotient.java:11)
```

See 14.1 Quotient.java

Example Using If

```
import java.util.Scanner;

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

check for potential exception

```
Enter two integers: 5 0 <Enter>
Divisor cannot be zero
```

See 14.2 QuotientWithIf.java

Example Using Method With Exit

```
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);
  }
}
```

check for potential exception

program terminates

```
Enter two integers: 5 3 <Enter>
5 / 3 is 1
```

```
Enter two integers: 5 0 <Enter>
Divisor cannot be zero
```

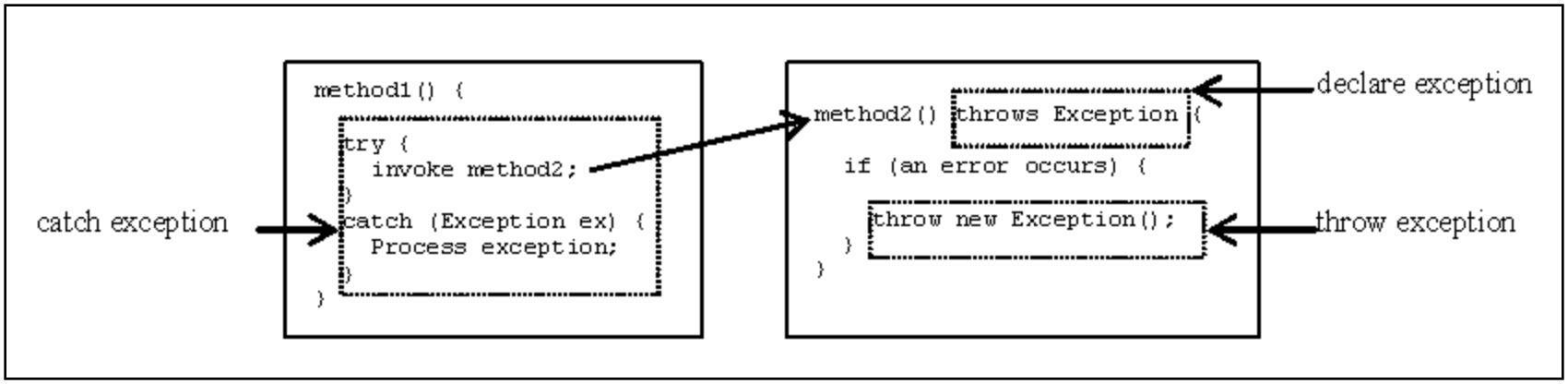
See 14.3 QuotientWithMethod.java

Exception Handling in Java

- **try** block contains code that is executed in normal circumstances
  - throws exception when necessary to avoid abnormal program termination
    - throwing an exception interrupts the normal flow of execution and passes execution from one place in the program to another
  - throw statement analogous to method call
    - calls catch block
- **catch** block contains code executed when exception of given type is thrown
  - catch block analogous to method definition with parameter type; however, program control **does not return** to throw statement
    - executes next statement after catch block

Exception Handling in Java

```
try {
   Code to try;
   Throw an exception with a throw statement or from method if necessary;
   More code to try;
}
catch (type ex) {
   Code to process the exception;
}
```



Example With Exception Handling

```
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 ...");
  }
}
```

calls method

See 14.4 QuotientWithException.java

Example With Exception Handling

```
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;
  }
```

throws exception looks for appropriate catch

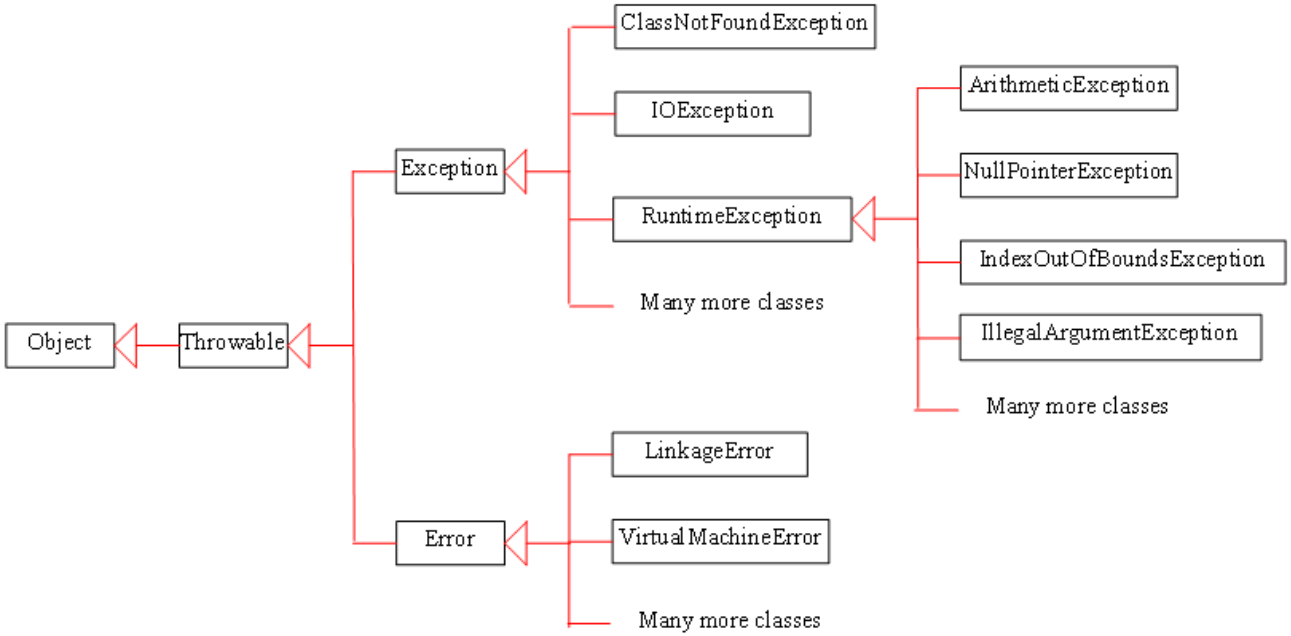
```
Enter two integers: 5 3 <Enter>
5 / 3 is 1
Execution continues ...
```

```
Enter two integers: 5 0 <Enter>
Exception: an integer cannot be divided by zero
Execution continues ...
```

See 14.4 QuotientWithException.java

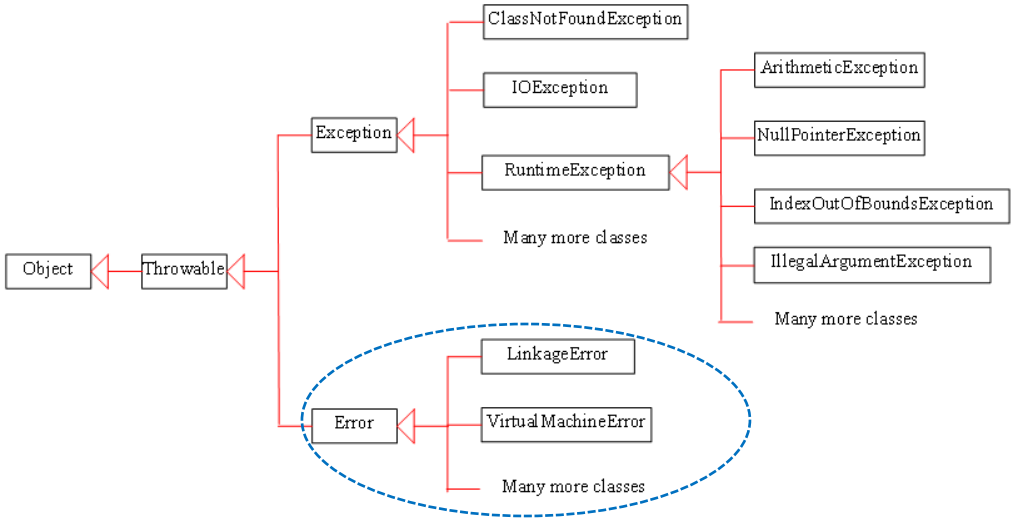
Exception Class Hierarchy

- Exceptions are objects; objects defined from classes
- Predefined exception classes from the Java API
- Throwable class is root of exception classes



Exception Types

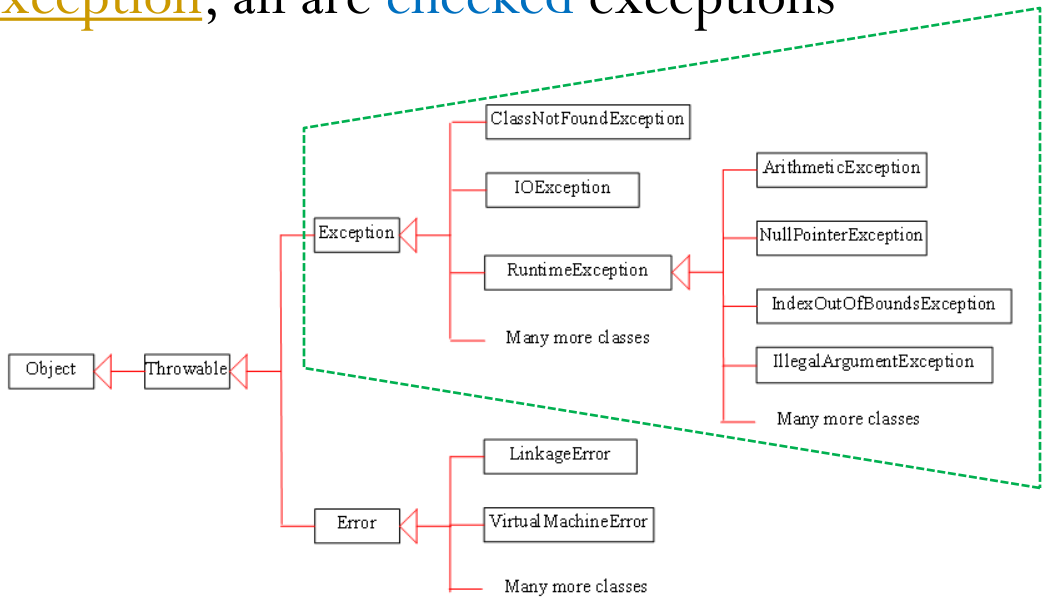
- System errors
  - Thrown by JVM
  - Represented by Error class (internal system errors)
  - Not much chance of recovery beyond notifying the user and terminating program gracefully (i.e. **LinkageError**, **VirtualMachineError**, etc.)
  - Unchecked exception



Exception Types

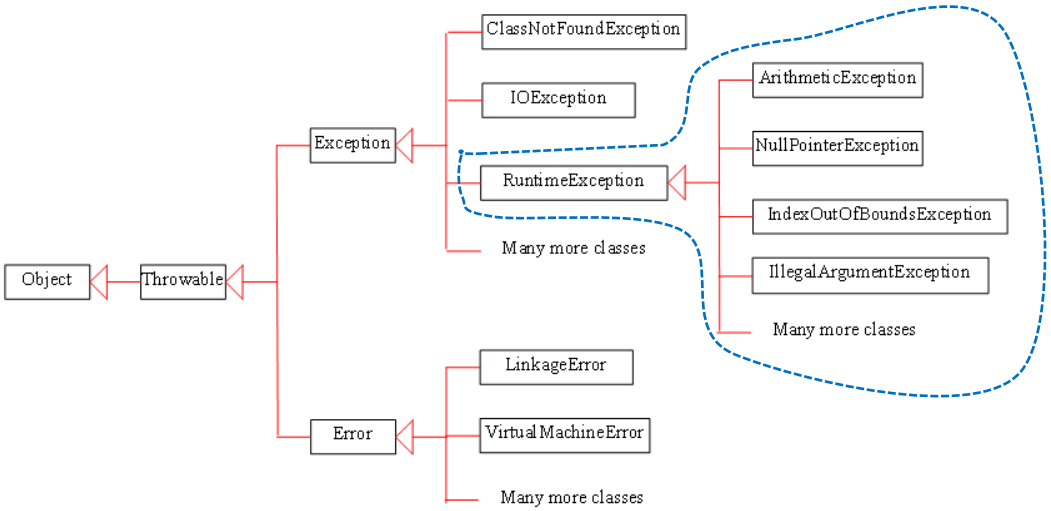
- Exceptions
  - Caused by your program and external circumstances
  - Represented by Exception class
  - Can be caught and handled by your program (i.e. **ClassNotFound**, **IOException**, etc.)
  - Except for RuntimeException, all are checked exceptions

Programmer must handle exception



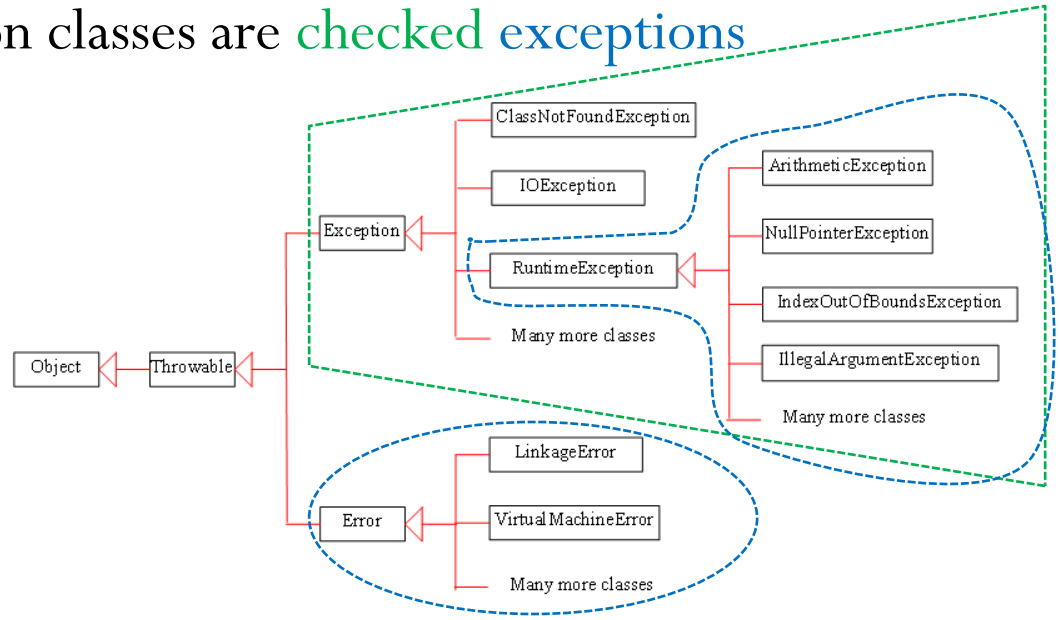
Exception Types

- Runtime exceptions
  - Caused by programming errors (bad casting, accessing out-of-bounds array, numeric errors, etc.)
  - Represented by RuntimeException class (i.e. **ArithmeticException**, **NullPointerException**, **IndexOutOfBoundsException**, etc.)
  - Generally thrown by JVM
  - Unchecked exception



Exception Types

- RuntimeException, Error, and their subclasses are unchecked exceptions
  - Compiler **does not force** programmer to deal with generated exceptions
  - In most cases they are unrecoverable logic errors
  - Can occur anywhere in a program
- All other exception classes are checked exceptions



Exception Types

- Compiler forces programmer to deal checked exceptions (other than **Error** or **RuntimeException**)
- If method declares checked exception, you **must** either
  - declare to **throw the exception** in the calling method
    - method body **not required** to catch exception
  - **catch exception** in a try-catch block, or

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

throws the exception

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

catch exception

Java Exception Handling Model

- Uses the following **three** operations
  - Declaring an exception
    - Every method **must** state the types of checked exception it might throw

```
public void myMethod() throws Exception1,
  Exception2, ..., ExceptionN
```

  - Throwing an exception
    - **Indirectly** through library method call
    - **Directly** by creating an instance of an appropriate exception type and throwing it

```
IllegalArgumentException ex = new
  IllegalArgumentException("Wrong Argument");
throw ex;
--OR--
throw new IllegalArgumentException("Wrong Argument");
```

See 14.7 CircleWithException.java

Java Exception Handling Model

- Uses the following **three** operations
  - Catching an exception
    - Can be caught and handled in a try-catch block

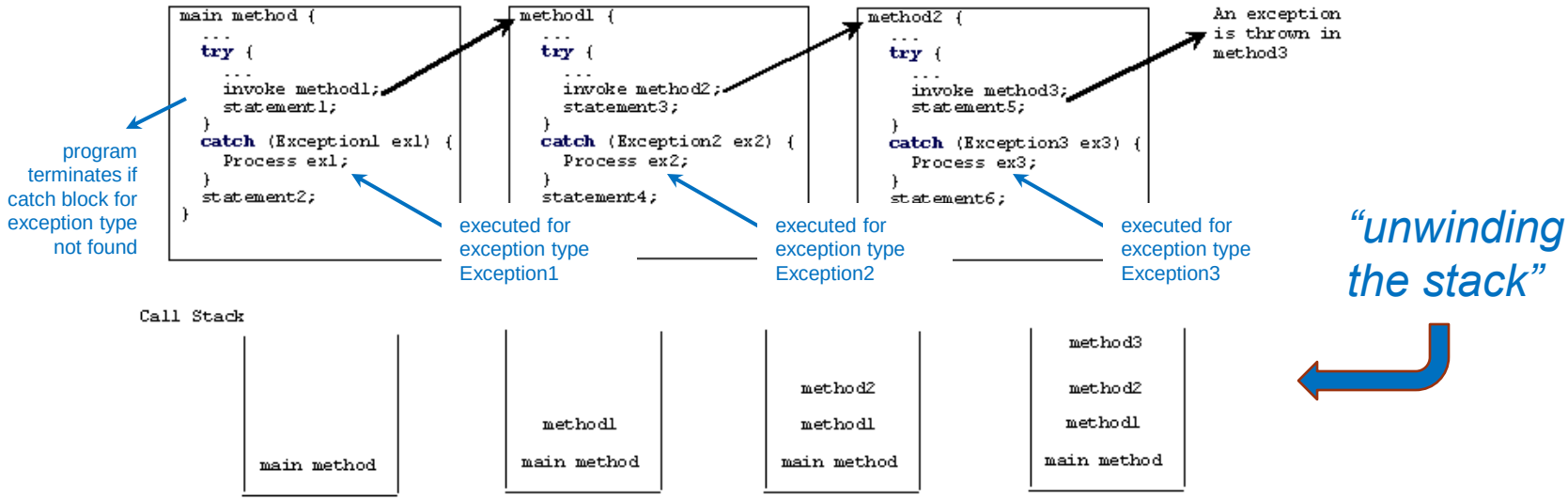
```
try {
   statements;
}
catch (Exception1 exVar1) {
   handler for exception1;
}
catch (Exception2 exVar2) {
   handler for exception2;
}
```

    - If no exceptions arise, catch blocks are skipped
    - Catching an exception → exception handler is found by propagating exception backward through method calls, starting from current method
      - Catch block is examined from first to last

See 14.8 TestCircleWithException.java

Java Exception Handling Model

- If exception is **not caught in current method**, it is passed to calling method
  - Process is repeated until
    - exception is caught **or**
    - run time error occurs and program terminates



Java Exception Handling Model

- If a catch block catches exceptions of a superclass, it can catch all the exception objects of the subclass
  - Compile error will results of a catch block for a superclass type appears before a catch block for a subclass type
    - Remember, catch blocks are examined from first to last!

```
try {
   . . .
}
catch (Exception ex) {
   . . .
}
catch (RuntimeException ex) {
   . . .
}
```

incorrect order

```
try {
   . . .
}
catch (RuntimeException ex) {
   . . .
}
catch (Exception ex) {
   . . .
}
```

correct order

specific to general

Getting Information from Exceptions

- Exception contains valuable information about the exception

java.lang.Throwable	
+getMessage(): String	Returns the message of this object.
+toString(): String	Returns the concatenation of three strings: (1) the full name of the exception class; (2) ":" (a colon and a space); (3) the getMessage() method.
+printStackTrace(): void	Prints the **Throwable** object and its call stack trace information on the console.
+getStackTrace(): StackTraceElement[]	Returns an array of stack trace elements representing the stack trace pertaining to this throwable.

→Example

Getting Information from Exceptions

```
java.lang.ArrayIndexOutOfBoundsException: 5
        at TestException.sum(TestException.java:24)
        at TestException.main(TestException.java:4)

5

java.lang.ArrayIndexOutOfBoundsException: 5

Trace Info Obtained from getStackTrace
method sum(TestException:24)
method main(TestException:4)
```

printStackTrace()

getMessage()

toString()

using getStackTrace()

See 14.6 TestException.java

The finally Clause

- Includes code that is executed regardless of whether an exception occurs or not
- catch block **may be omitted** when finally clause is used
- Executes even if return statement prior to finally block

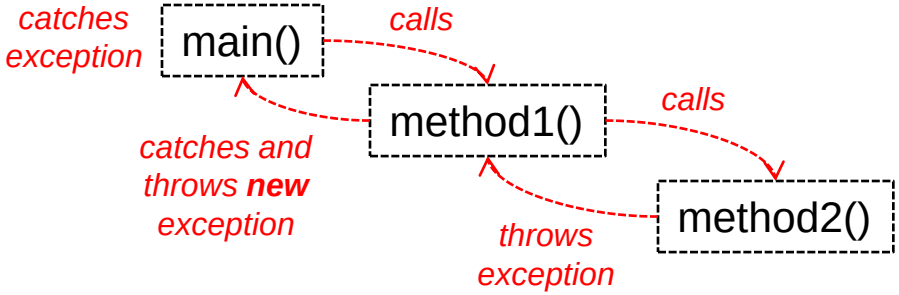
```
try {
  exceptionGeneratingStatements;
}
catch (TheException ex) {
   caughtExceptionStatements;
}
finally {
  finalStatements;
}
afterTryBlockStatements;
```

Possible Case	catch block executed	finally block executed	statements after try block executed
No exception		Y	Y
Exception generated and caught	Y	Y	Y
Exception generated but not caught		Y *(exception passed to calling method)*	

More on Exceptions

- Rethrowing Exceptions
  - If handler cannot process exception, or calling method should be notified of exception
    - **Rethrow exception** so other handlers get a chance to process exception
- Chained Exceptions
  - Throw a **new exception** (with additional information) along with original exception

```
try {
   statements;
}
catch (TheException ex) {
   // perform operations before
    // rethrowing exception;
   throw ex;
}
```



See 14.9 ChainedExceptionDemo.java

Creating Custom Exception Classes

- If existing exception classes cannot adequately represent your problem, create custom exception class derived from Exception or from subclass of Exception
  - Remember that **all** methods from Throwable class are inherited

```
public class InvalidRadiusException extends Exception {
   private double radius;
   /** Construct an exception */
   public InvalidRadiusException(double radius) {
      super("Invalid radius " + radius);
      this.radius = radius;
   }
   /** Return the radius */
   public double getRadius() {
      return radius;
   }
}
```

Most exception classes in Java API contain two constructors: no-arg and String parameter.

See 14.10 InvalidRadiusException.java

Using Custom Exception Classes

```
class CircleWithCustomException {
  /** The radius of the circle */
  private double radius;

  /** The number of the objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithCustomException () throws InvalidRadiusException {
    this(1.0);
  }

  /** Construct a circle with a specified radius */
  public CircleWithCustomException (double newRadius) throws InvalidRadiusException {
    setRadius(newRadius);
    numberOfObjects++;
  }

  /** Return radius */
  public double getRadius() {
    return radius;
  }
```

Programmer must handle checked exception

See 14.11 TestCircleWithCustomException.java

Using Custom Exception Classes

```
  /** Set a new radius */
  public void setRadius(double newRadius) throws InvalidRadiusException {
    if (newRadius >= 0)
      radius =  newRadius;
    else
      throw new InvalidRadiusException(newRadius);
  }

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  public double findArea() {
    return radius * radius * 3.14159;
  }
}
```

Programmer must handle checked exception

See 14.11 TestCircleWithCustomException.java

Using Custom Exception Classes

```
public class TestCircleWithCustomException {
   public static void main(String[] args) {
      try {
         new CircleWithCustomException(5);
         new CircleWithCustomException(-5);
         new CircleWithCustomException(0);
      }
       catch (InvalidRadiusException ex) {
          System.out.println(ex);
       }
     System.out.println("Number of objects created: " +
       CircleWithCustomException.getNumberOfObjects());
   }
}
```

Programmer must handle checked exception

See 14.11 TestCircleWithCustomException.java

File Class

- Contains methods for getting properties of file/directory, renaming, and deleting a file/directory
- Provides abstraction for handling machine dependent complexities of files and path names in a machine-independent manner
  - File name is a string that is passed to constructor
  - File class is **wrapper** for filename and directory path
- Absolute file name contains file name with complete path and drive letter

  `c:\book\Welcome.java`
  `/home/liang/book/Welcome.java`
- Relative file name contains file name relative to current directory

  `image/us.gif` *(Java directory separator same as Unix)*
- **Does not contain** methods for creating a file or for reading/writing data from/to a file

File Class

```
new File("c:\\book")
→ creates File object for
   directory book

new File("c:\\book\\test.dat")
→ creates File object for file
   c:\book\test.dat
```

→Example

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 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.
+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 its parent directories if the parent directories do not exist.

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()));
  }
}
```

See 14.12 TestFileClass.java

File Class Example

Windows

```
Command Prompt
C:\book>java TestFileClass
Does it exist? true
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
What is its absolute path? C:\book\.\image\us.gif
What is its canonical path? C:\book\image\us.gif
What is its name? us.gif
What is its path? .\image\us.gif
When was it last modified? Sat May 08 14:00:34 EDT 1999
What is the path separator? ;
What is the name separator? \

C:\book>
```

Unix

```
Command Prompt - telnet panda
$ pwd
/home/liang/book
$ java TestFileClass
Does it exist? true
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
What is its absolute path? /home/liang/book/./image/us.gif
What is its canonical path? /home/liang/book/image/us.gif
What is its name? us.gif
What is its path? ./image/us.gif
When was it last modified? Wed Jan 23 11:00:14 EST 2002
What is the path separator? :
What is the name separator? /
$
```

See 14.12 TestFileClass.java

PrintWriter Class

- Used to create a file and write data to a text file

```
PrintWriter output = new PrintWriter(new File(filename));
```

  - Possible to throw checked exception with constructor
    - FileNotFoundException
- Can use `print`, `println`, and `printf` ** methods on `PrintWriter` object to write data to a file

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."

***verify '\n' formatting*

→Example

PrintWriter Class Example

```
public class WriteData {
  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);
    }

    // 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();
  }
}
```

PrintWriter constructor may throw checked I/O exception

See 14.13 WriteData.java

Scanner Class

- Used to read strings and primitive values from the console
  - Breaks input into tokens delimited by whitespace characters

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

  - Possible to throw checked exception with constructor
    - FileNotFoundException

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.

Scanner Class

- Token-reading methods (**`nextByte()`**, **`nextShort()`**, **`nextInt()`**, **`nextLong()`** **`nextFloat()`**, **`nextDouble()`** **and** **`next()`**) read tokens separated by delimiters
  - Default delimiters are whitespace
  - Skips delimiters, reads token ending at delimiter
    - Does **not** read delimiter after token
    - **`next()`** reads string delimited by delimiters; **`nextLine()`** reads line ending with line separator
      - This is why using **`nextLine()`** after a token reading delimiter will return an empty line!
      - Characters read from delimiter to line separator
        - Data on separate lines of console input will store an empty line if **`nextLine()`** is used after token reading delimiter
  - Token is converted into appropriate value type

→Example

Scanner Class Example

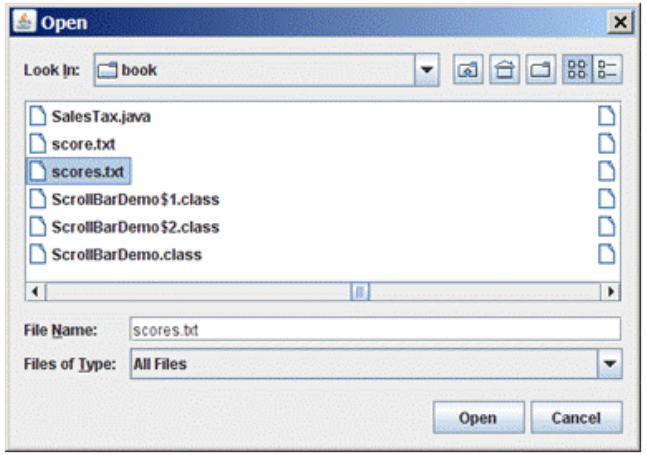
```
import java.util.Scanner;

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();
  }
}
```

Scanner constructor may throw checked I/O exception

See 14.14 ReadData.java

GUI File Dialogs

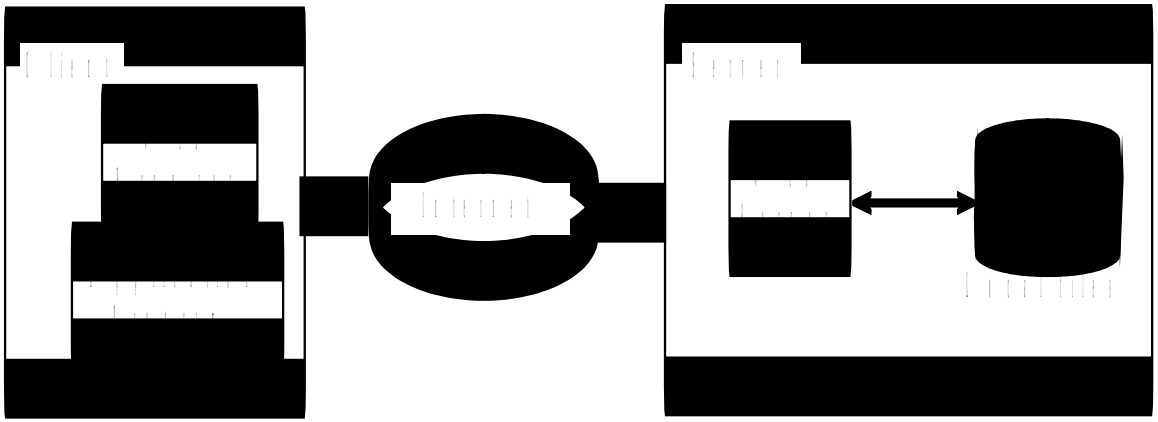
- `JFileChooser` class displays file dialog for choosing a file
- Methods
  - showOpenDialog → displays dialog box
    - Returns `int` value (APPROVE_OPTION or CANCEL_OPTION)
  - getSelectedFile → returns selected file as a file object



See 14.16 ReadFileUsingJFileChooser.java

Reading Data from the Web

- Read data from file on Web similar to reading data from file on computer
- URL (Uniform Resource Locator) is unique address for Web file



→Example

Reading Data from the Web

URL constructor may throw checked MalformedURLException exception

- Create URL object

```
try {
    URL url = new URL("www.google.com/index.html");
}
catch (MalformedURLException ex) {
    ex.printStackTrace();
}
```

- Use openStream() method to open an input stream to create a Scanner object as follows:

```
Scanner input = new Scanner(url.openStream());
```

- Read data from stream same as from local file

```
String line = input.nextLine();
```

See 14.17 ReadFileFromURL.java