Chapter 12 Exception Handling and Text IO

Exception-Handling Overview

Show runtime error

Quotient Run

Fix it using an if statement

QuotientWithIf Run

With a method

QuotientWithMethod Run

Runtime Error?

```
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));
  }
}
```

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

Fix it Using an if Statement

```
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 ");
  }
}
```

Suppose there is another method that can throw the exception

```
3  public class QuotientWithMethod {
4    public static int quotient(int number1, int number2) {
5      if (number2 == 0) {
6        System.out.println("Divisor cannot be zero");
7        System.exit(1);
8      }
9
10     return number1 / number2;
11   }
12
13   public static void main(String[] args) {
14     Scanner input = new Scanner(System.in);
15
16     // Prompt the user to enter two integers
17     System.out.print("Enter two integers: ");
18     int number1 = input.nextInt();
19     int number2 = input.nextInt();
20
21     int result = quotient(number1, number2);
22     System.out.println(number1 + " / " + number2 + " is "
23       + result);
24   }
25 }
```

Exception Advantages

QuotientWithException | Run

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

```
3  public class QuotientWithException {
4    public static int quotient(int number1, int number2) {
5      if (number2 == 0)
6        throw new ArithmeticException("Divisor cannot be zero");
7
8      return number1 / number2;
9    }
10
11   public static void main(String[] args) {
12     Scanner input = new Scanner(System.in);
13
14     // Prompt the user to enter two integers
15     System.out.print("Enter two integers: ");
16     int number1 = input.nextInt();
17     int number2 = input.nextInt();
18
19     try {
20       int result = quotient(number1, number2);
21       System.out.println(number1 + " / " + number2 + " is "
22         + result);
23     }
24     catch (ArithmeticException ex) {
25       System.out.println("Exception: an integer " +
26         "cannot be divided by zero ");
27     }
28
29     System.out.println("Execution continues ...");
30   }
31 }
```

If an ArithmeticException occurs

Handling InputMismatchException

InputMismatchExceptionDemo Run

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

Handling an exception and continuing program execution

```
3  public class InputMismatchExceptionDemo {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      boolean continueInput = true;
7
8      do {
9        try {
10         System.out.print("Enter an integer: ");
11         int number = input.nextInt();
12
13         // Display the result
14         System.out.println(
15           "The number entered is " + number);
16
17         continueInput = false;
18       }
19       catch (InputMismatchException ex) {
20         System.out.println("Try again. (" +
21           "Incorrect input: an integer is required)");
22         input.nextLine(); // Discard input
23       }
24     } while (continueInput);
25   }
26 }
```

If an InputMismatchException occurs

The File Class

The File class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.

The filename is a string.

The File class is a wrapper class for the file name and its directory path.

File class

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.

File class

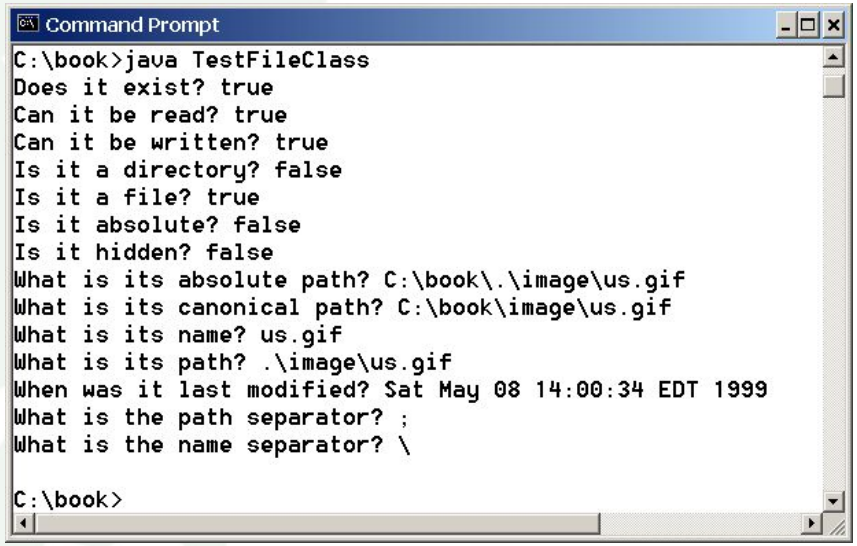
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), 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.

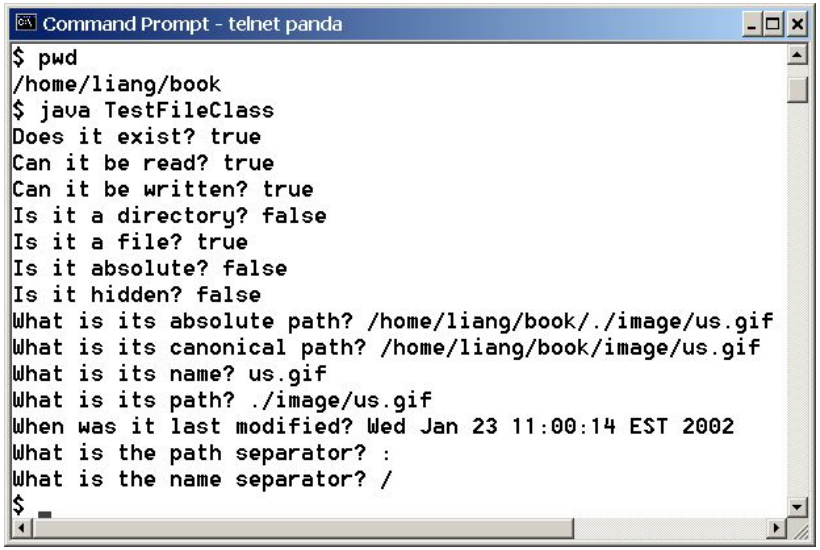
Text I/O

- 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.
- This section introduces how to read/write strings and numeric values from/to a text file using the **Scanner** and **PrintWriter** classes.

Problem: Explore File Properties

Objective: Write a program that demonstrates how to create files in a platform-independent way and use the methods in the File class to obtain their properties. The following figures show a sample run of the program on Windows and on Unix.





TestFileClass Run

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()));
  }
}
```

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 §4.6, "Formatting Console Output and Strings."

WriteData Run

Example

```
File file = new File("scores.txt");
if (file.exists()) {
  System.out.println("File already exists");
  System.exit(0);
}


// Create a file
PrintWriter output = new 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();
```

Try-with-resources

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;

}

```
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); }
```

WriteDataWithAutoClose

Run

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.

ReadData Run

Example

```
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();
  }
}
```

Problem: Replacing Text

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
```

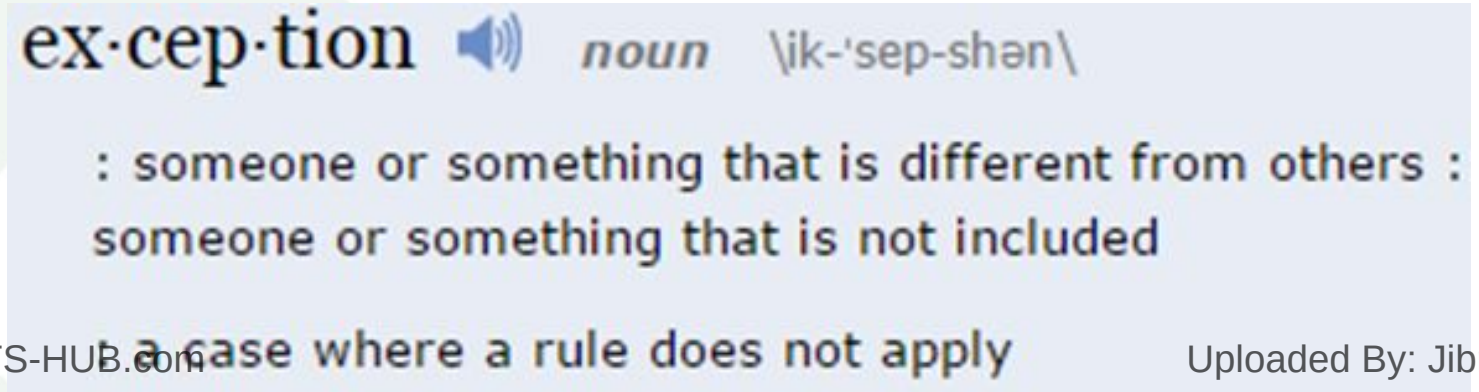
replaces all the occurrences of StringBuilder by StringBuffer in FormatString.java and saves the new file in t.txt.

ReplaceText Run

```
// Check command line parameter usage
if (args.length != 4) {
  System.out.println(
    "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
  System.exit(1);
}
File sourceFile = new File(args[0]);
if (!sourceFile.exists()) {
  System.out.println("Source file " + args[0] + " does not exist");
  System.exit(2);
}
File targetFile = new File(args[1]);
if (targetFile.exists()) {
  System.out.println("Target file " + args[1] + " already exists");
  System.exit(3);
}
try (
  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);
  }
}
```

Exception Handling

- Exception handling technique enables a method to **throw** an exception to its caller.
- Without this capability, a method must handle the exception or terminate the program.

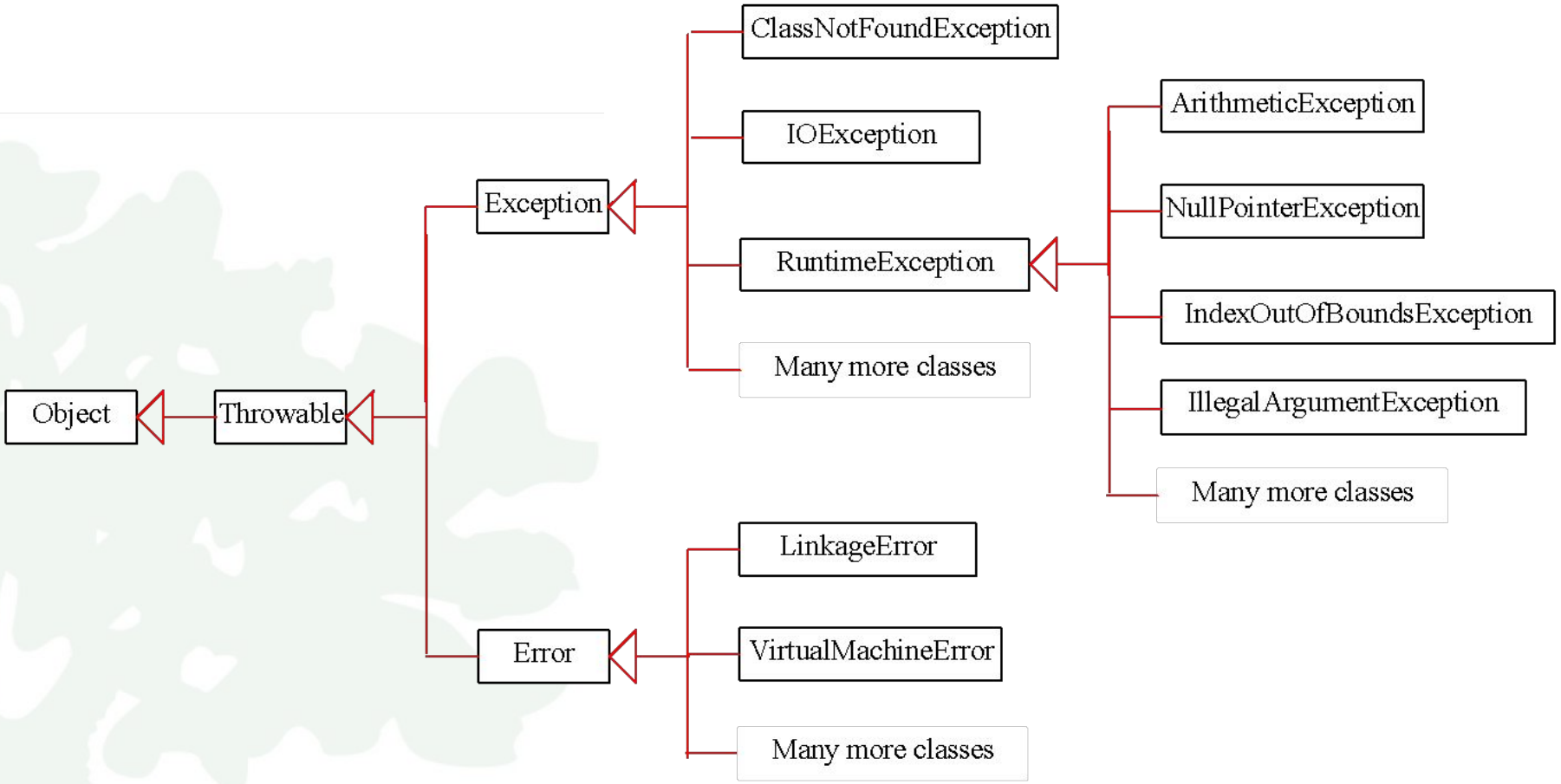


What's the Output?

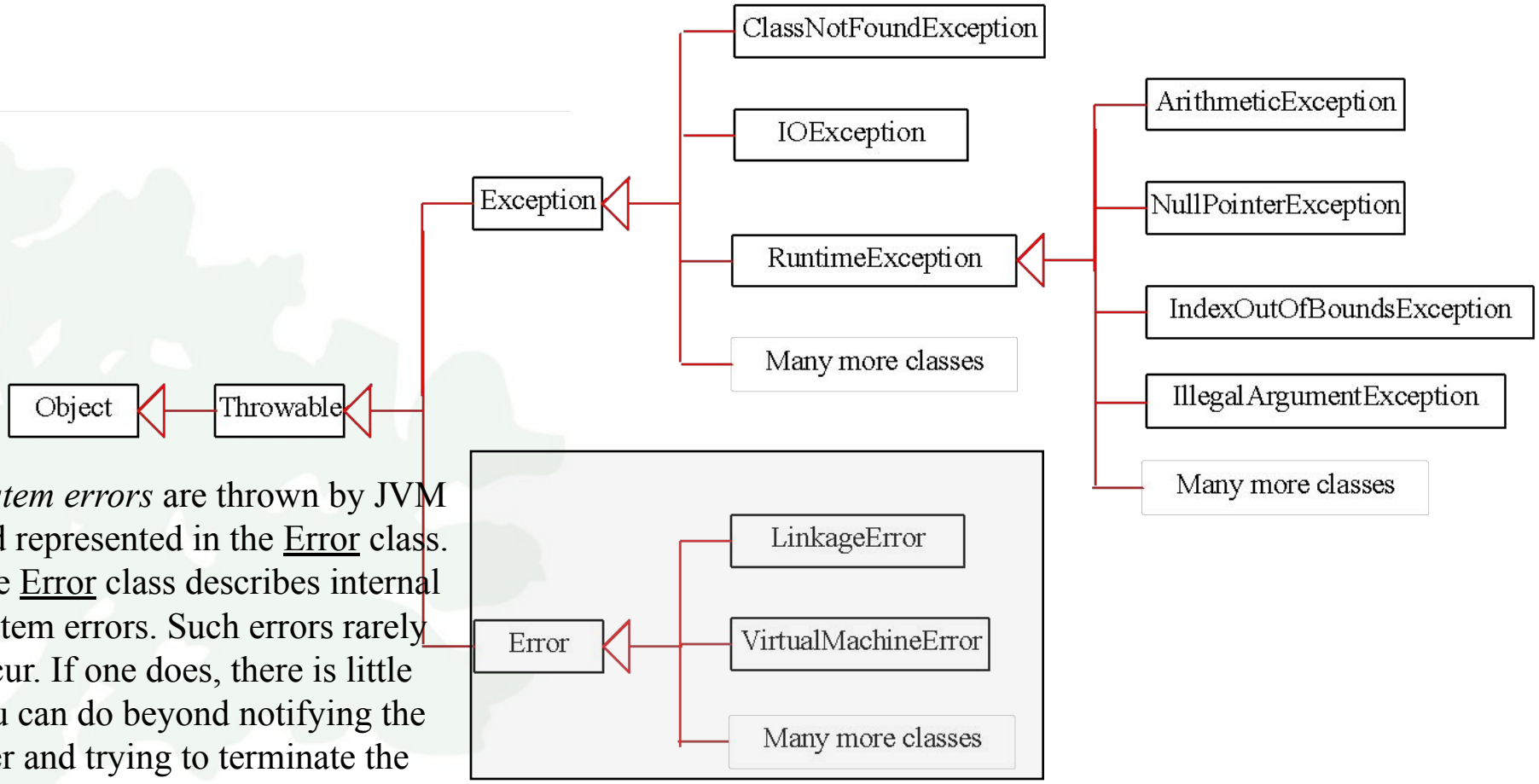
```
public class Test {
  public static void main(String[] args) {
    for (int i = 0; i < 2; i++) {
      System.out.print(i + " ");
      try {
        System.out.println(1 / 0);
      }
      catch (Exception ex) {
      }
    }
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    try {
      for (int i = 0; i < 2; i++) {
        System.out.print(i + " ");
        System.out.println(1 / 0);
      }
    }
    catch (Exception ex) {
    }
  }
}
```

Exception Types



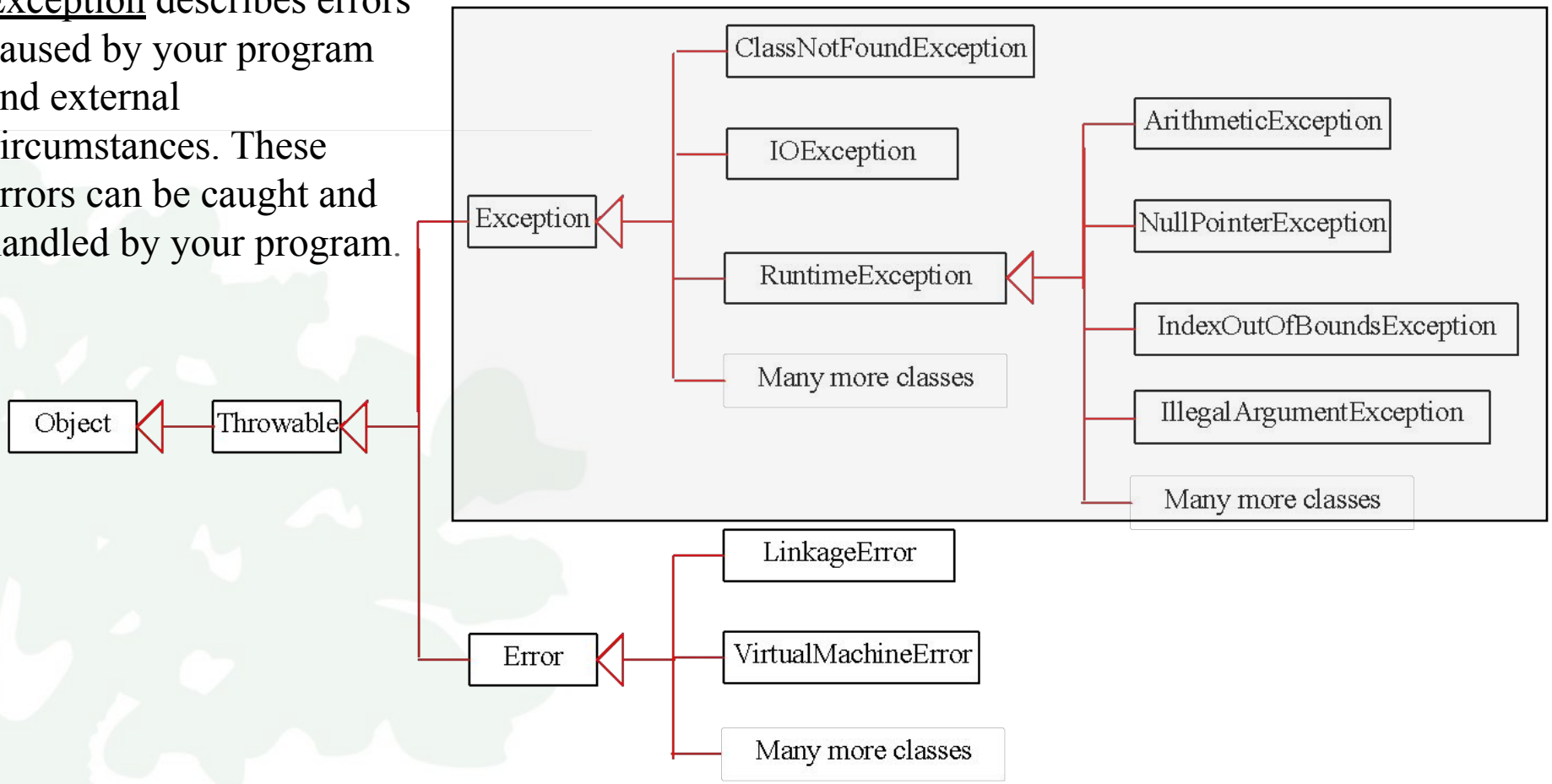
System Errors



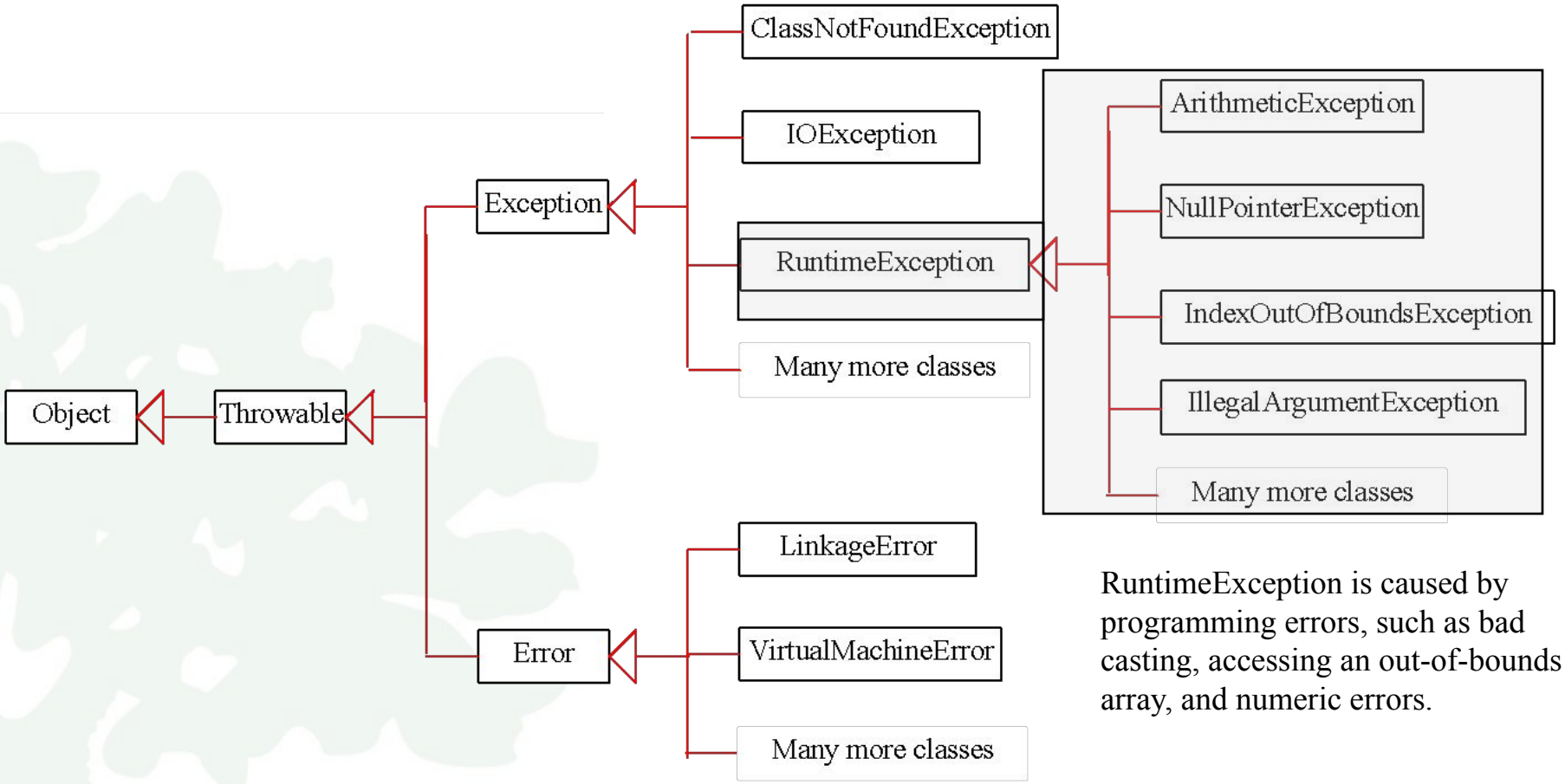
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

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



Runtime Exceptions



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

What is the Exception?!

```java
public class Test {
  public static void main(String[] args) {
    System.out.println(1 / 0);
  }
}
```

(a)

```java
public class Test {
  public static void main(String[] args) {
    int[] list = new int[5];
    System.out.println(list[5]);
  }
}
```

(b)

```java
public class Test {
  public static void main(String[] args) {
    String s = "abc";
    System.out.println(s.charAt(3));
  }
}
```

(c)

```java
public class Test {
  public static void main(String[] args) {
    Object o = new Object();
    String d = (String)o;
  }
}
```

(d)

```java
public class Test {
  public static void main(String[] args) {
    Object o = null;
    System.out.println(o.toString());
  }
}
```

(e)

```java
public class Test {
  public static void main(String[] args) {
    System.out.println(1.0 / 0);
  }
}
```

(f)

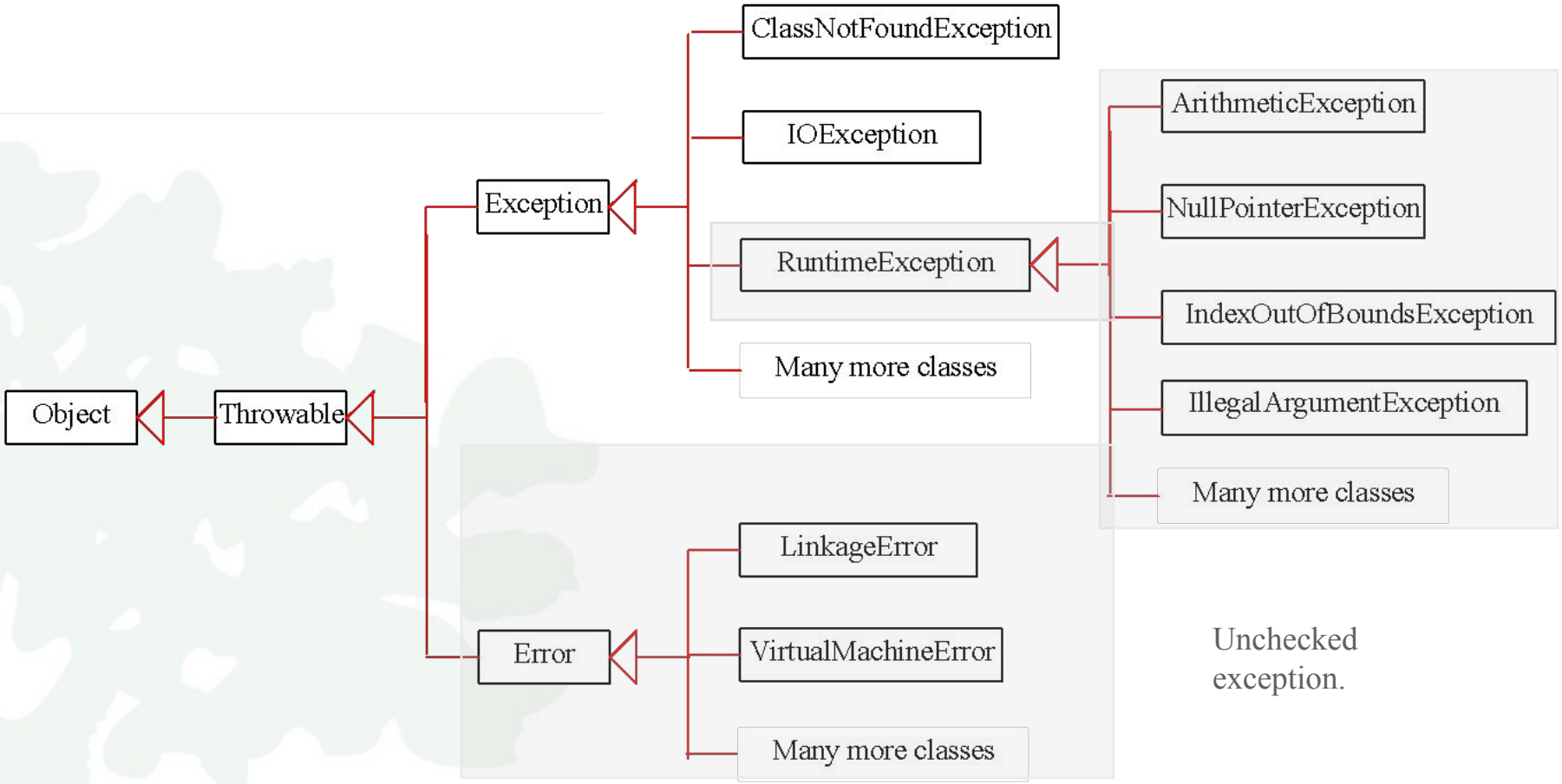
Checked Exceptions vs. Unchecked Exceptions

- **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.

Unchecked Exceptions

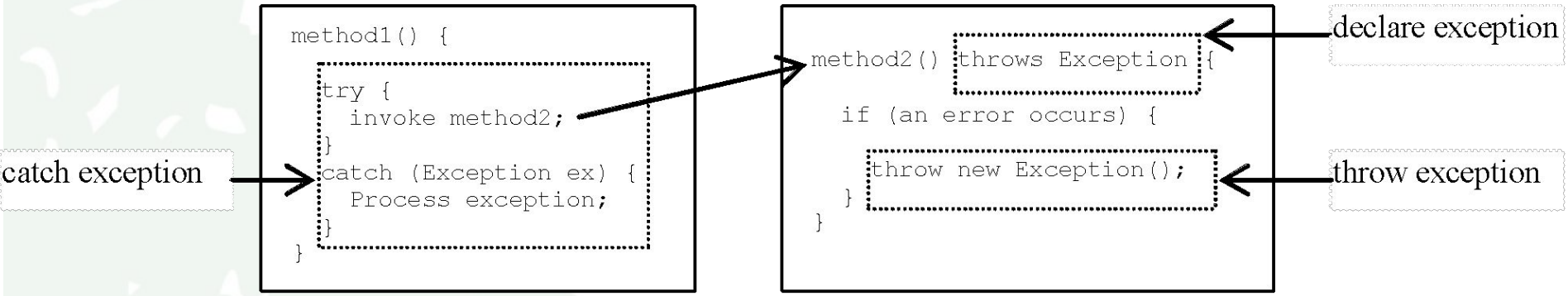
- In most cases, unchecked exceptions reflect programming **logic errors** that are not recoverable.
- For example:
  - a **NullPointerException** is thrown if you access an object through a reference variable before an object is assigned to it.
  - an **IndexOutOfBoundsException** is thrown if you access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

Unchecked Exceptions



Unchecked exception.

Declaring, Throwing, and Catching Exceptions



Declaring Exceptions

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

```
public void myMethod()
  throws IOException

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

Throwing Exceptions

- When the program detects an error, the program can create an **instance** of an appropriate exception type and throw it.
- This is known as **throwing an exception**.

```
throw new TheException();

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

Throwing Exceptions Example

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

Catching Exceptions

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

Catch or Declare Checked Exceptions

- Java forces you to deal with checked exceptions.
- You must invoke it in a **try-catch** block **or** declare to **throw** the exception in the calling method.
- For example, suppose that method **p1** invokes method **p2** and **p2** may throw a checked exception (e.g., **IOException**), you have to write the code as follow:

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

(a)

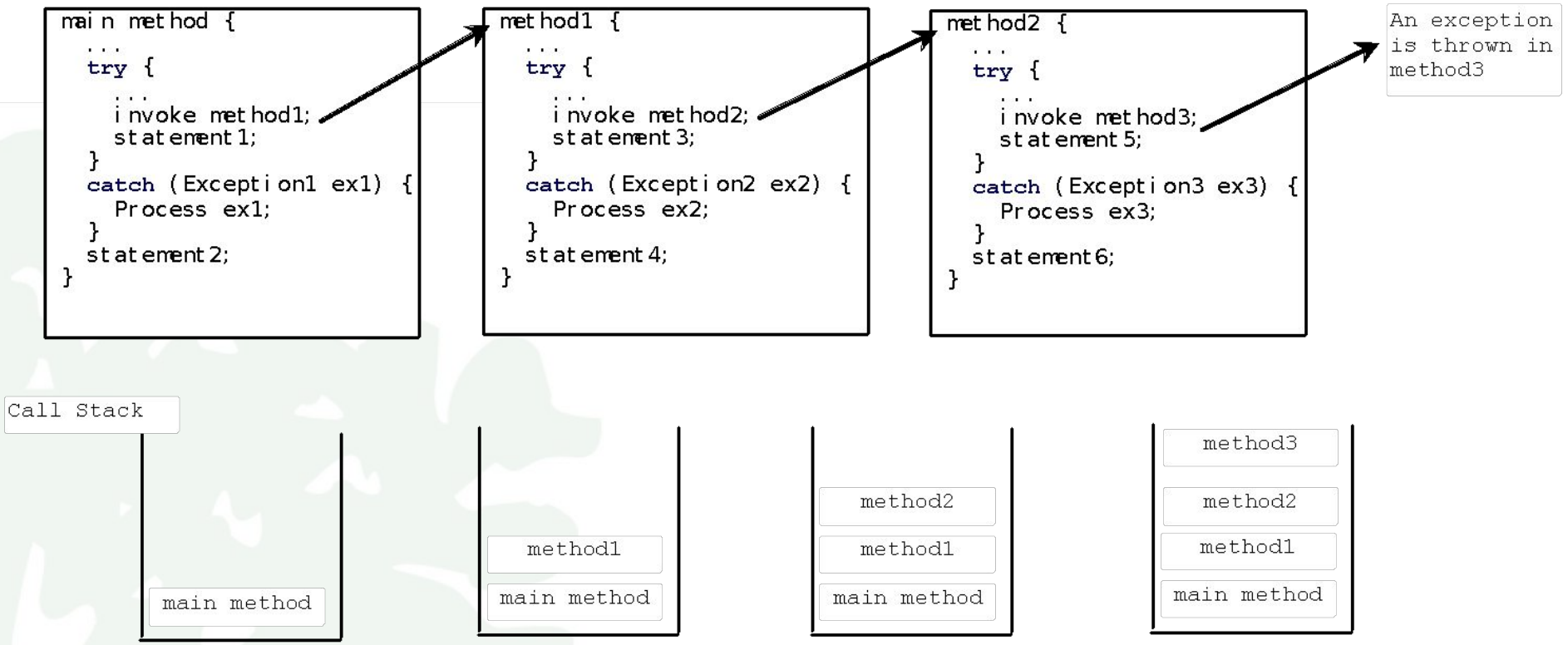
```
void p1() throws IOException {

  p2();

}
```

(b)

Catching Exceptions



Catch or Declare Checked Exceptions

Suppose p2 is defined as follows:

```
void p2() throws IOException {
  if (a file does not exist) {
     throw new IOException("File does not exist");
  }

  ...
}
```

Catch or Declare Checked Exceptions

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

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

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

Example: Declaring, Throwing, and Catching Exceptions

- Objective: This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in Chapter 9. The new setRadius method throws an exception if radius is negative.

CircleWithException

TestCircleWithException Run

```
1  public class CircleWithException {
2    /** The radius of the circle */
3    private double radius;
4
5    /** The number of the objects created */
6    private static int numberOfObjects = 0;
7
8    /** Construct a circle with radius 1 */
9    public CircleWithException() {
10     this(1.0);
11   }
12
13   /** Construct a circle with a specified radius */
14   public CircleWithException(double newRadius) {
15     setRadius(newRadius);
16     numberOfObjects++;
17   }
18
19   /** Return radius */
20   public double getRadius() {
21     return radius;
22   }
```

throws IllegalArgumentExcention

```
24    /** Set a new radius */
25    public void setRadius(double newRadius)
26        throws IllegalArgumentException {
27      if (newRadius >= 0)
28        radius =  newRadius;
29      else
30        throw new IllegalArgumentException(
31          "Radius cannot be negative");
32    }
33
34    /** Return numberOfObjects */
35    public static int getNumberOfObjects() {
36      return numberOfObjects;
37    }
38
39    /** Return the area of this circle */
40    public double findArea() {
41      return radius * radius * 3.14159;
42    }
43  }
```

```
1  public class TestCircleWithException {
2    public static void main(String[] args) {
3      try {
4        CircleWithException c1 = new CircleWithException(5);
5        CircleWithException c2 = new CircleWithException(-5);
6        CircleWithException c3 = new CircleWithException(0);
7      }
8      catch (IllegalArgumentException ex) {
9        System.out.println(ex);
10     }
11
12     System.out.println("Number of objects created: " +
13       CircleWithException.getNumberOfObjects());
14   }
15 }
```

What's the Exception?

```
public class Test {
  public static void main(String[] args) {
    try {
      int[] list = new int[10];
      System.out.println("list[10] is " + list[10]);
    }
    catch (ArithmeticException ex) {
      System.out.println("ArithmeticException");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException");
    }
    catch (Exception ex) {
      System.out.println("Exception");
    }
  }
}
```

What's the Exception?

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (ArithmeticException ex) {
      System.out.println("ArithmeticException");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException");
    }
    catch (Exception e) {
      System.out.println("Exception");
    }
  }

  static void method() throws Exception {
    System.out.println(1 / 0);
  }
}
```

What's the Exception

```
public class Test {
  public static void main(String[] args) {
    try {
      method();
      System.out.println("After the method call");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in main");
    }
    catch (Exception ex) {
      System.out.println("Exception in main");
    }
  }

  static void method() throws Exception {
    try {
      String s ="abc";
      System.out.println(s.charAt(3));
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException in method()");
    }
    catch (Exception ex) {
      System.out.println("Exception in method()");
    }
  }
}
```