

Exceptions and Exception Handling

Three Types of Programming Errors

1. Syntax Errors

- Source code (e.g. Java program) does not follow the syntax rules for the language.
- Caught by the compiler (e.g. javac)

2. Logic Errors

- Algorithm(s) defined or implemented incorrectly (program compiles, but is incorrect)
- Caught using testing, repaired through ‘debugging’

3. Runtime Errors (“Exceptions”)

During execution, program requests an operation that is impossible to carry out.

Examples of Runtime Errors (Exceptions)

- **Invalid input**
- **Attempt to open file that doesn't exist**
- **Network connection broken**
- **Array index out of bounds**

Method Call Stack and Stack Trace

Method Call Stack (“Call Stack”)

- The stack that records data associated with the **chain of method calls leading to the current method being executed**

Stack Trace: A record of method invocations

- Summary of call stack contents
- Listed from most (top) to least (bottom) recent method. `main()` normally at the bottom of the method call stack
 - **source line numbers** for statements that invoke a method and the last statement executed in the current method are given.
- **If an exception is not caught**, a Java program will display the exception and stack trace, and then stop (e.g. `ExceptionDemo.java`)
 - The first (active) method will have thrown the exception

Catching and Handling Exceptions

Catching Exceptions

When a runtime error occurs, program given the **state of execution (stack trace)**, and the **type of error**

Exception 'Handlers'

Code executed when exceptions are caught; allow a program to recover from and/or repair the problem (rather than stop execution)

Catching Exceptions Using try-catch

```
try {  
    // statements that might throw an exception  
    statement1;  
    statement2;
```

If exception occurs here, jump out of try block before next instruction

```
}  
catch (Exception1 e1) {  
    // handler for Exception1
```

```
}  
catch (Exception2 e2) {  
    // handler for Exception2
```

```
}  
...  
catch {ExceptionN eN} {  
    // handler for ExceptionN
```

```
}
```

```
// Statements after try-catch  
nextStatement;
```

If exception occurred, search list of 'catch' statements for **first matching exception** type & execute associated handler. Then execute first statement after catch blocks. **(NOTE: exceptions must be listed from most to least specific class because of search order!)**

If no 'catch' matches the exception, the exception is **passed back to the calling method**, current method is exited.

Catching and Handling Exceptions Continued

Try-Catch Block Structure

- Define a scope for a set of commands that **may produce exceptions** ('try'), and
- A subsequent **list of exception handlers** ('catch' statements) to invoke when exceptions are 'thrown' (occur)

Control Flow in a 'try-catch' Block

- When an exception occurs in a 'try' block, **execution jumps to the end of the 'try' block (the end brace '}')**.
- Java then searches the list of 'catch' statements in order, selecting the first matching handler
 - (see: HandleExceptionDemo.java)

Addition to try-catch: the *finally* clause (try-catch-finally)

Purpose

- Define a block of code that will execute *regardless* of whether an exception is caught or not for a try block
(executes after try and catch blocks)
- Finally block will execute even if a return statement precedes it in a try block or catch block (!)

Example Uses

I/O programming: ensure that a file is always closed.

Define error-handling code needed for different errors in one place within a method.

FinallyDemo.java (from text)

```
public class FinallyDemo {  
    public static void main(String[] args) {  
        java.io.PrintWriter output = null;  
  
        try {  
            // Create a file  
            output = new java.io.PrintWriter("text.txt");  
  
            // Write formatted output to the file  
            output.println("Welcome to Java");  
        }  
        catch (java.io.IOException ex) {  
            ex.printStackTrace();  
        }  
        finally {  
            // Close the file  
            if (output != null) output.close();  
        }  
    }  
}
```

Declaring, Throwing, and Catching Exceptions

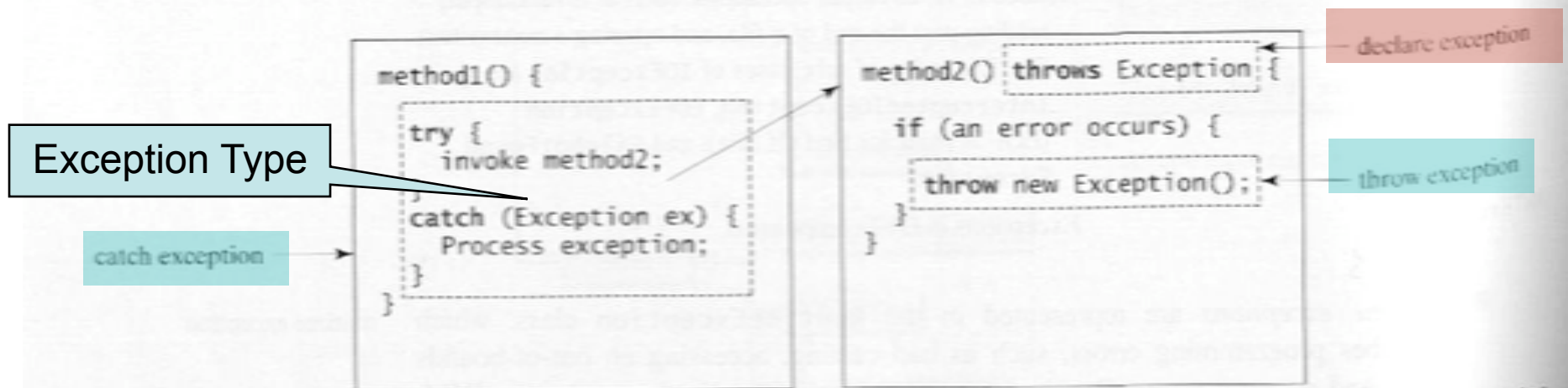


FIGURE 17.4 Exception handling in Java consists of declaring exceptions, throwing exceptions, and catching and processing exceptions.

- “Throwing” an exception means to use the “throw” command to generate a message (an object that is a subclass of Exception)
- “Declaring” an exception means to add it to a **list of (checked) exceptions** at the end of a method signature, e.g.

```
public void myMethod() throws Exception1, ..., ExceptionN { ... }
```

this list must contain all checked exceptions that the method may throw. - 11 -

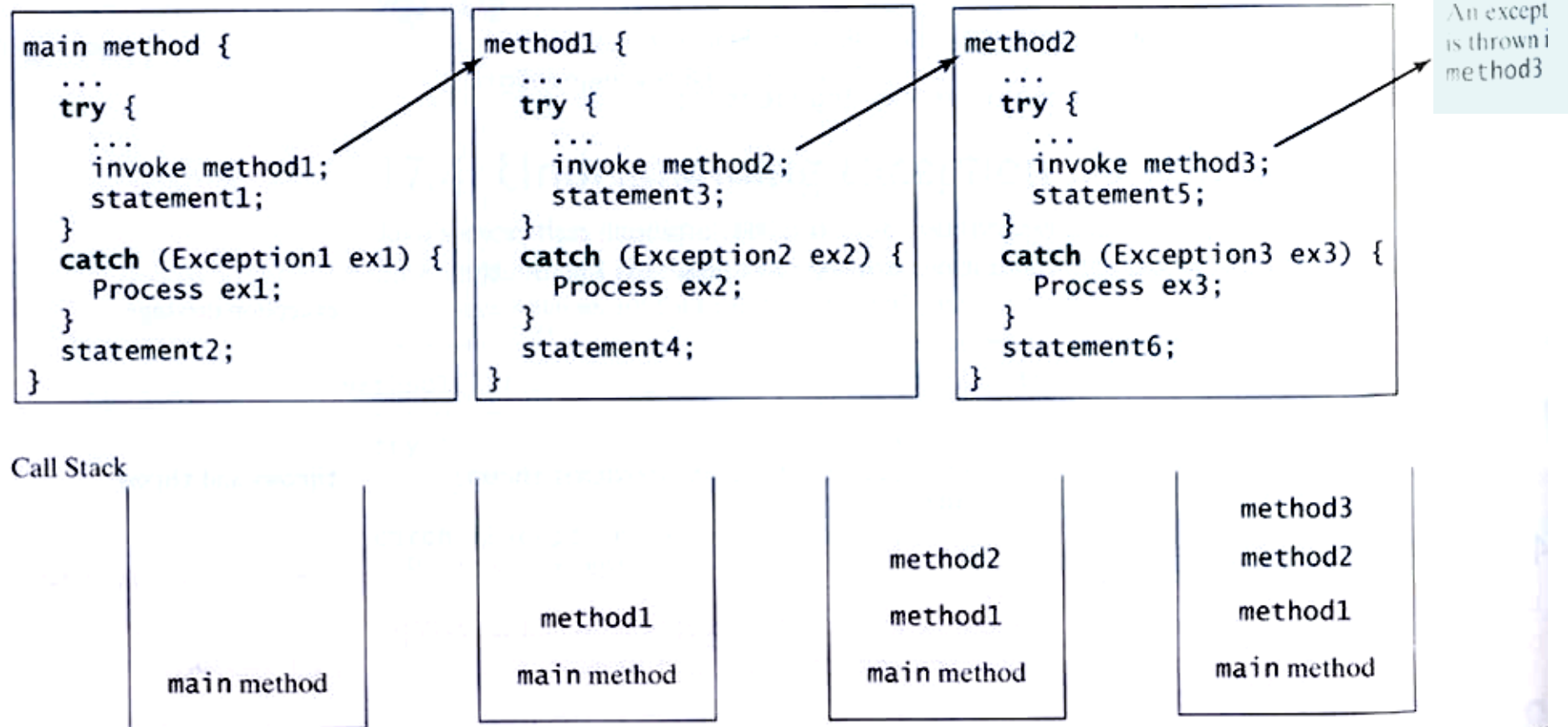


FIGURE 17.5 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the main method.

Cases to consider: in method2, throwing Exception3, Exception2, Exception1, SomeOtherException objects (each type being a subclass of 'Exception')

Getting Information from Exceptions

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

FIGURE 17.6 **Throwable** is the root class for all exception objects.

Example: TestException.java (p. 586)

(The *message* is a text string associated with the Throwable object (e.g. exception))

Exception (Runtime Error) Types in Java

System Errors (*Error*)

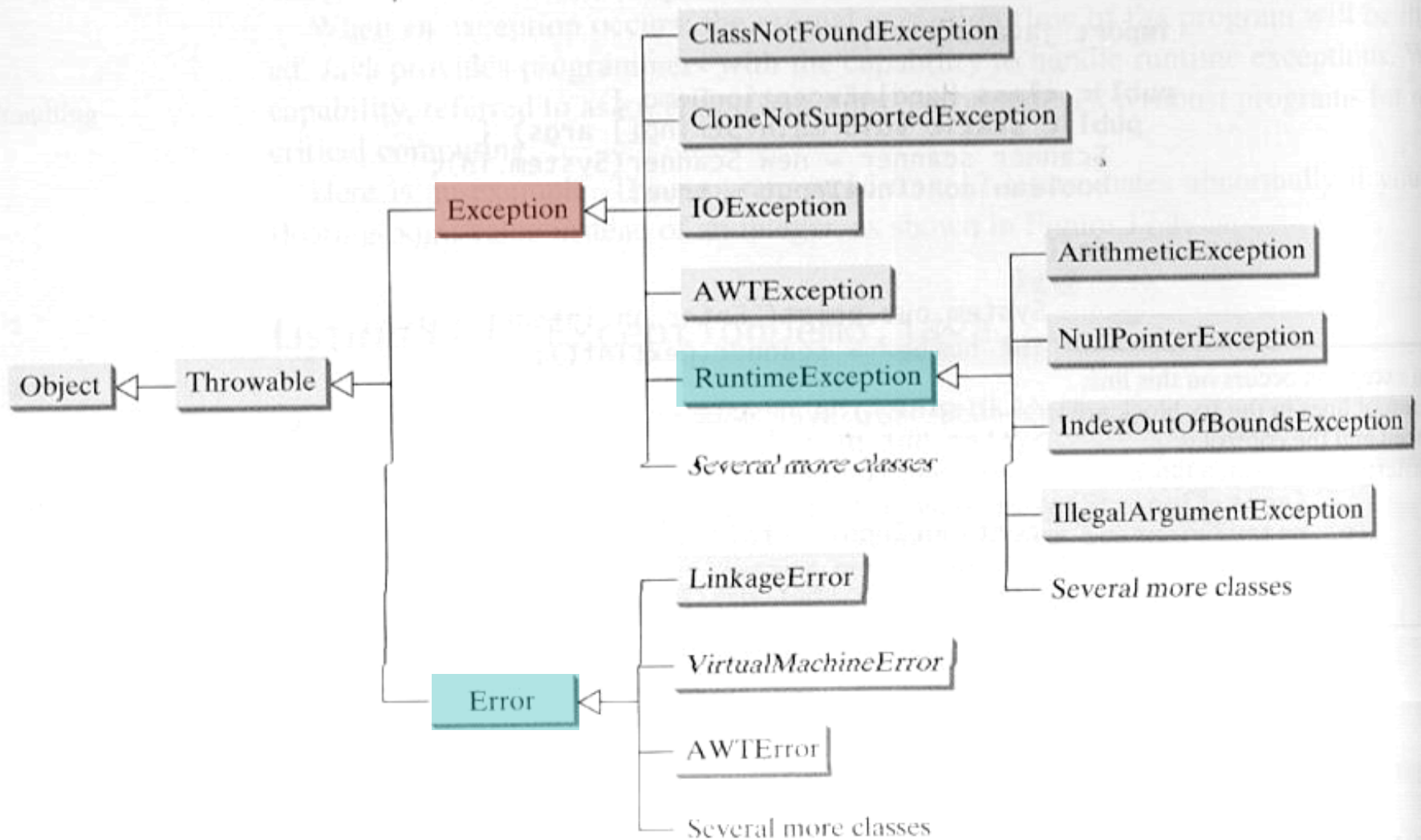
- Thrown by the Java Virtual Machine (JVM)
- Internal system errors (rare), such as incompatibility between class files, JVM failures

Runtime Exceptions (*RuntimeException*)

- Also normally thrown by JVM
- Usually **unrecoverable programming errors** (e.g. divide by 0, array index error, null reference)

(“Normal”) Exceptions (*Exception*)

- Errors that may be caught and handled (e.g. file not found)



Unchecked and Checked Exceptions

Unchecked Exceptions

- Error, RuntimeException, and subclasses
- These exceptions are normally not recoverable (cannot be handled usefully, e.g. NullPointerException)
- The javac compiler **does not force these exceptions to be declared or caught** (to keep programs shorter), but they can be.

Checked Exceptions

- Exception class and subclasses (excluding RuntimeException)
- Compiler forces the programmer to catch and handle these.

Defining New Exception Classes

Java Exception Classes

Are numerous; use these where possible.

New Exception Classes

Are derived from Exception or a subclass of Exception.

Constructors for Exception Classes

Constructors are normally either no-arg, or one argument
(takes the string message as an argument)


```
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;  
    }  
}
```

Example Use:

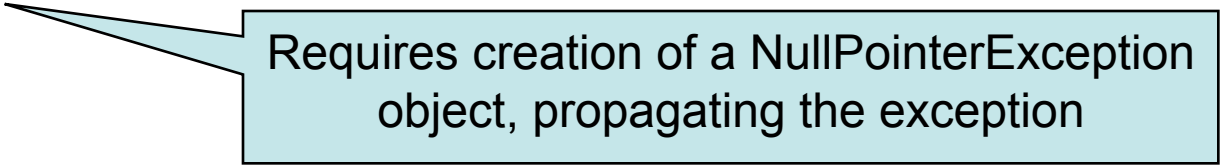
```
    if ( x <= 0 )  
        throw new InvalidRadiusException(x);
```

When Do I Use Exceptions?

(Course text) “The point is not to abuse exception handling as a way to deal with a simple logic test.”

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

vs.



Requires creation of a NullPointerException object, propagating the exception

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

Use exceptions for ‘unexpected’ errors (unusual situations). Simple errors specific to a method should be handled within the method (locally), as above.