

Exception Handling



CS2012: Introduction to Programming II

Exception Handling

- ▶ ***exception handling***: allows a program to deal with "exceptional" situations (runtime errors) and continue its normal execution.
 - your program can actually recover from a runtime error and continue without crashing.
- ▶ ***runtime errors***: errors which occur while a program is running if the JVM finds an operation that is impossible for it to carry out.
 - **ArrayIndexOutOfBoundsException**: triggered when an array index is out of bounds.

Exception Handling

- ▶ Java "throws" runtime errors as exceptions
- ▶ ***exception:***
 - an object that represents an error or condition that prevents execution from proceeding normally.
- ▶ ***unhandled exception:***
 - an exception that has not been dealt with.
 - causes the program to crash.

Exception-Handling Example

Exception-Handling Example

- ▶ exceptions are throw from a method, the calling method can catch and handle the exception.

```
1  import java.util.Scanner;
2
3  public class Quotient {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in);
6
7          // Prompt the user to enter two integers
8          System.out.print("Enter two integers: ");
9          int number1 = input.nextInt();
10         int number2 = input.nextInt();
11
12         System.out.println(number1 + " / " + number2 + " is " +
13             (number1 / number2));
14     }
15 }
```

Exception-Handling Example

► Error Checking Method 1: Using an If Statement.

```
1  import java.util.Scanner;
2
3  public class QuotientWithIf {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in);
6
7          // Prompt the user to enter two integers
8          System.out.print("Enter two integers: ");
9          int number1 = input.nextInt();
10         int number2 = input.nextInt();
11
12         if (number2 != 0)
13             System.out.println(number1 + " / " + number2
14                                 + " is " + (number1 / number2));
15         else
16             System.out.println("Divisor cannot be zero ");
17     }
18 }
```

Exception-Handling Example

► Error Checking Method 2: Using a Separate Method.

```
1  import java.util.Scanner;
2
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-Handling Example

► Error Checking Method 3: Using Exception-Handling

```
1  import java.util.Scanner;
2
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 Arithmetic Exception occurs

Throwing and Catching Exceptions

Throwing an Exception

- ▶ Line 6: `throw new ArithmeticException("Divisor cannot be zero")`
 - notice the **throw** keyword, indicates that an exception of the type `ArithmeticException` should be "thrown"
 - thrown when `number2` (the denominator) is 0
 - `ArithmeticException`, is a Class type in Java which is used to indicate a runtime error due to an invalid math operation.
 - this is called "throwing an exception"

Throwing an Exception

- ▶ thrown exceptions interrupt normal flow of execution
- ▶ "throwing an exception" passes the exception from one place in the program to another.
- ▶ most "thrown" exceptions have to be dealt with somewhere and your program will crash if they are not dealt with.
 - so if you "throw an exception", then at some point you need to "catch" that exception.

Catching an Exception

```
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 ");  
}
```

Catching an Exception

- ▶ Whenever you invoke a method which has the potential to throw an checked exception, then you **MUST** surround the invocation of the method in a try/catch structure.
- ▶ `try` contains the code that is executed under normal conditions
- ▶ `catch` is only executed if the code from the `try` block triggers (throws) an exception
 - this is where the exception is handled
- ▶ Whether or not an exception is thrown, the code after the catch clause is executed as the program continues.

Catching an Exception

- ▶ The **catch** clause must identify the type of exception it is catching:

```
catch (ArithmeticException ex)
```

- NOTE: the value of the thrown exception can be accessed through the parameter of the catch (ex)

- ▶ try-catch block template:

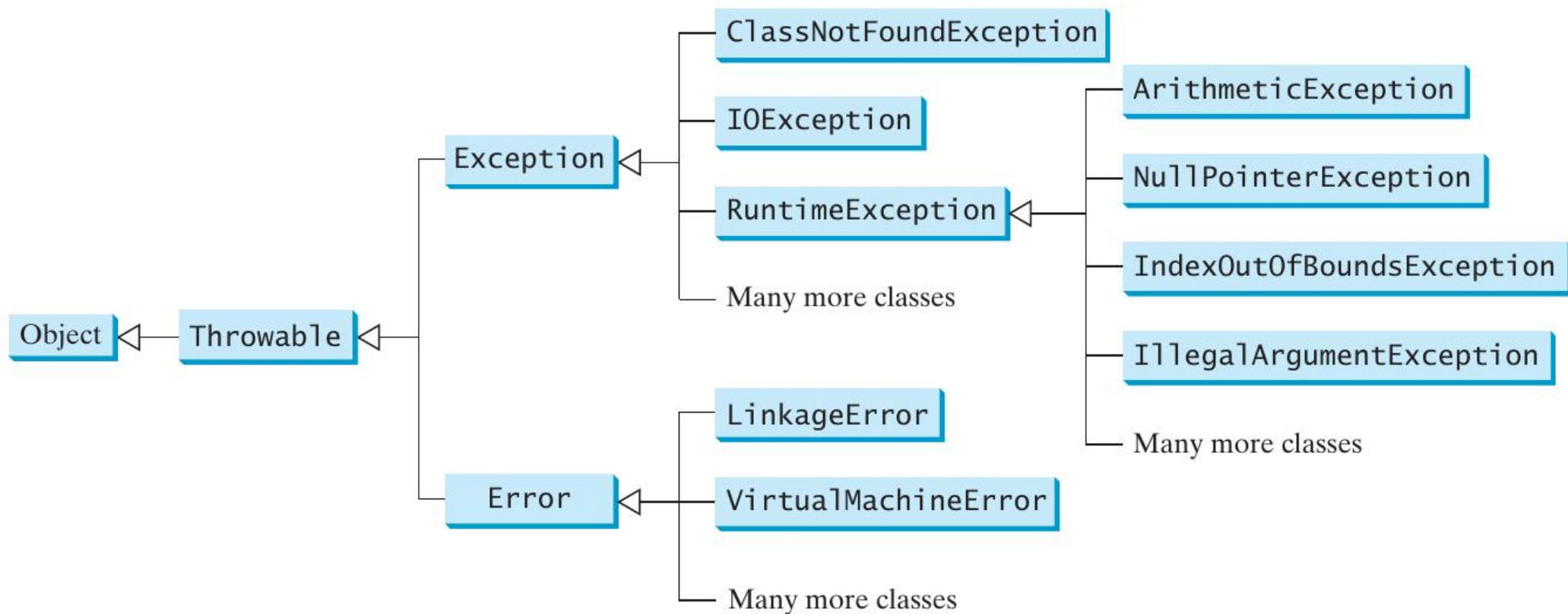
```
try {  
    Code to run;  
    A statement or a method that may throw an exception;  
    More code to run;  
}  
catch (type ex) {  
    Code to process the exception;  
}
```

Catching an Exception

- ▶ exceptions can be thrown directly in a try block, or by invoking a method that may throw an exception.
- ▶ most of the time a called method does not know how to handle an error, and it is up to the caller, to deal with the error.

Exception Types

Exception Types



Exception Types

- ▶ Throwable is root of all exceptions
 - all exceptions inherit from Throwable
 - custom exceptions can be created by extending Exception or one of its subclasses.
- ▶ Exception classes are classified in three major types:
 - System Errors
 - Exceptions
 - Runtime Exceptions

System Errors

- ▶ thrown by the JVM
- ▶ represented by Error Class
 - describes internal system errors
 - these errors rarely occur
- ▶ if a system error occurs, and you know about it, then all you can do is notify the user and try to terminate the program gracefully.
- ▶ these would be errors that Oracle would have to fix with a Java update.

TABLE 12.1 Examples of Subclasses of **Error**

<i>Class</i>	<i>Reasons for Exception</i>
LinkageError	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
VirtualMachineError	The JVM is broken or has run out of the resources it needs in order to continue operating.

Exception Errors

- ▶ represented by the `Exception` class
- ▶ errors caused by the program, or external circumstances
- ▶ can be caught and handled easily by your program

TABLE 12.2 Examples of Subclasses of `Exception`

<i>Class</i>	<i>Reasons for Exception</i>
<code>ClassNotFoundException</code>	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the <code>java</code> command, or if your program were composed of, say, three class files, only two of which could be found.
<code>IOException</code>	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of <code>IOException</code> are <code>InterruptedException</code> , <code>EOFException</code> (EOF is short for End of File), and <code>FileNotFoundException</code> .

Runtime Exceptions

- ▶ represented by the `RuntimeException` class
- ▶ describes programming errors (bad casting, out of bounds index, numeric errors, etc)
- ▶ generally thrown by the JVM.

TABLE 12.3 Examples of Subclasses of `RuntimeException`

<i>Class</i>	<i>Reasons for Exception</i>
<code>ArithmeticException</code>	Dividing an integer by zero. Note that floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
<code>NullPointerException</code>	Attempt to access an object through a <code>null</code> reference variable.
<code>IndexOutOfBoundsException</code>	Index to an array is out of range.
<code>IllegalArgumentException</code>	A method is passed an argument that is illegal or inappropriate.

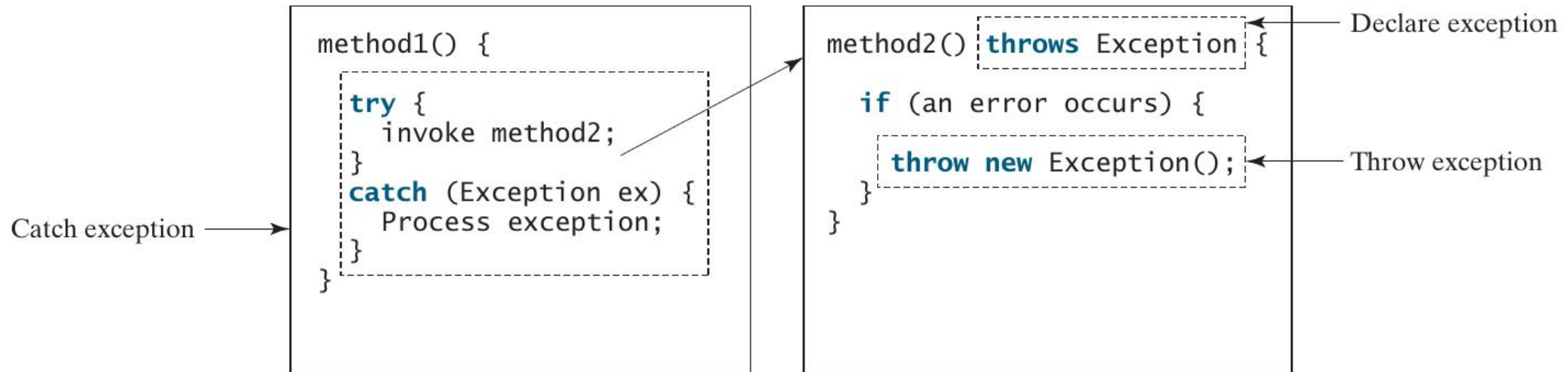
Checked vs. Unchecked Exceptions

- ▶ RuntimeException, Error, and any subclasses of these are known as ***unchecked exceptions***
 - the compiler does not force you to check and deal with them in a try-catch block
 - Example: NullPointerException, IndexOutOfBoundsException
 - can occur anywhere in a program and are unchecked to avoid overuse of try-catch blocks.
- ▶ All other exceptions are ***checked exceptions***
 - the compiler forces you to deal with these

Exception Handling Process

Finding a Handler for an Exception

- ▶ when an exception is thrown, a handler for that exception is found by tracing backward through the chain of method calls leading up to the exception.



- ▶ exception-handling is based on three operations:
 - declaring an exception
 - throwing an exception
 - catching an exception

Declaring Exceptions

▶ *declaring exceptions:*

- every method must declare the types of checked exceptions that it might throw
- Remember: Error and RuntimeException types do not have to be declared because they are unchecked

▶ to declare an exception use the **throws** keyword in the method header

```
public void myMethod() throws ExceptionType
```

▶ if a method throws more than one exception list them all separated by commas:

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

▶ NOTE: if a superclass method does not declare an exception, you cannot override it to declare one in the subclass.

Throwing Exceptions

- ▶ errors can be created as an instance of the appropriate exception type and ***thrown***
- ▶ Example: program detects a method argument violates the valid types of arguments to a method i.e. an argument cannot be negative:

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

- ▶ Or you can throw an anonymous exception:

```
throw new IllegalArgumentException("Wrong Argument");
```

Throwing Exceptions

- ▶ NOTE: Most exceptions in the Java API have two constructors:
 - a no-arg constructor
 - a constructor which takes a String argument that describes the nature of the error.
 - ◆ this is called the ***exception message*** and can be accessed using `getMessage()`

- ▶ TIP:
 - to declare an exception use `throws`
 - to throw an exception use `throw`

Catching Exceptions

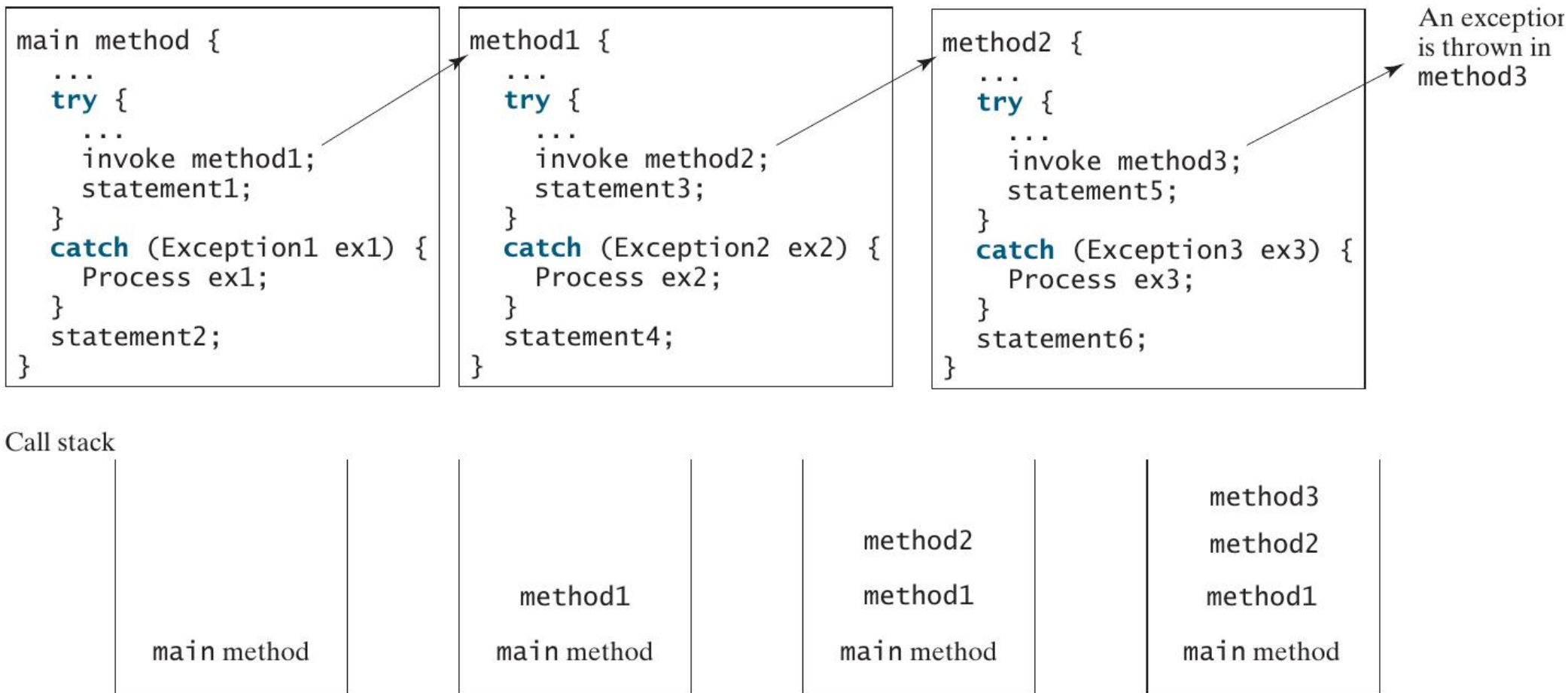
- ▶ Thrown exceptions can be caught and handled in try-catch blocks:

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

Catching Exceptions

- ▶ if no exceptions trigger in the `try` block, all catch blocks are skipped
- ▶ if any statement from `try` throws an exception:
 - Java skips all remaining statements in the `try` block
 - Java tries to find the code to handle the exception
 - method calls are "propagated backward" until an **exception handler** is found
 - each catch block is tried from first to last
 - ◆ if the exception type matches, then that catch block is executed
 - if no handler is found java keeps passing the exception up the chain of method calls until one is found or the program terminates if nothing is found.

Catching Exceptions: An Example



Catching Exceptions

- ▶ Note: Exceptions classes can be derived from a common superclass.
- ▶ If a catch block catches an exception of a superclass type, it can catch all the exception objects derived from that superclass.

Catching Exceptions

- ▶ The order of the exceptions in a catch block is important
- ▶ compile errors occur if a catch block for a superclass type appears before a catch block for a subclass type.

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

(a) Wrong order

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

(b) Correct order

Catching Exceptions

- ▶ Note: you MUST deal with checked exceptions which means you have to invoke a method which throws a checked exception in a try-catch block or declare to throw the exception somewhere else

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

(a) Catch exception

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

(b) Throw exception

Multi-Catch

- ▶ JDK 7 introduced a feature called the multi-catch to simplify coding for exceptions with the same handling code.
- ▶ Each exceptions is seperated by the pipe character | (think half of an OR operator)

```
catch (Exception1 | Exception2 | ... | Exceptionk ex) {  
    // Same code for handling these exceptions  
}
```

Retrieving Exception Information

- ▶ The following methods from Throwable, allow you to get information about the thrown exception

java.lang.Throwable

+getMessage(): String

+toString(): String

+printStackTrace(): void

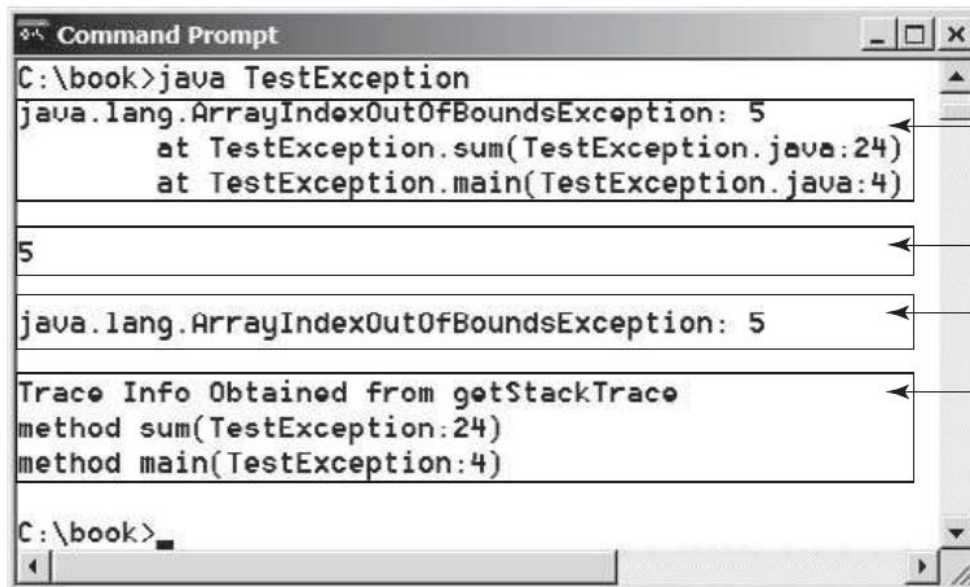
+getStackTrace():
StackTraceElement[]

Returns the message that describes this exception object.

Returns the concatenation of three strings: (1) the full name of the exception class; (2) ":" (a colon and a space); (3) the getMessage() method.

Prints the Throwable object and its call stack trace information on the console.

Returns an array of stack trace elements representing the stack trace pertaining to this exception object.



```
Command Prompt
C:\book>java TestException
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)

C:\book>
```

printStackTrace()

getMessage()

toString()

Using getStackTrace()

The Finally Clause

The finally Clause

- ▶ An optional clause added to try/catch

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

The finally Clause

- ▶ The code in `finally` is executed under all circumstances, regardless of an exception occurring
 - if no exception in `try`, `finalStatements` executed and next statement after `try` is executed.
 - if exception is thrown and caught by the same catch as the `try` block, rest of the statements in `try` are skipped, catch block is executed, and `finally` is executed, then statements after `try` are executed.
 - if an exception is thrown and caught somewhere else, statements in `try` are skipped, `finally` is executed, exception is passed to the caller of the method.
- ▶ `finally` is executed even if there is a `return` statement prior to reaching the `finally` block

Other Exception-Handling Details

Rethrowing Exceptions

- ▶ an exception handler can "rethrow" an exception if the handler cannot process the exception or just wants to let the caller be notified of the exception

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

Chained Exceptions

- ▶ **chained exception:** throwing a new exception along with another exception
- ▶ usually you would throw a new exception with additional information along with the original

```
1 public class ChainedExceptionDemo {
2     public static void main(String[] args) {
3         try {
4             method1();
5         }
6         catch (Exception ex) {
7             ex.printStackTrace();
8         }
9     }
10
11     public static void method1() throws Exception {
12         try {
13             method2();
14         }
15         catch (Exception ex) {
16             throw new Exception("New info from method1", ex);
17         }
18     }
19
20     public static void method2() throws Exception {
21         throw new Exception("New info from method2");
22     }
23 }
```

```
java.lang.Exception: New info from method1
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
    at ChainedExceptionDemo.main(ChainedExceptionDemo.java:4)
Caused by: java.lang.Exception: New info from method2
    at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
    at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:13)
    ... 1 more
```

Custom Exception Classes

Custom Exception Classes

- ▶ Java gives many exception classes, use them whenever possible.
- ▶ however, if you can't apply an existing exception to an error in your program you can write your own.
- ▶ all exception should be derived from Exception or one of its subclasses.
 - If you extend from RuntimeException, your class will be an ***unchecked*** exception.
 - If you extend from Exception, your class will be a ***check*** exception.
- ▶ Custom exceptions are always classes by themselves

Defining Custom Exception Classes

```
1  public class InvalidRadiusException extends Exception {
2      private double radius;
3
4      /** Construct an exception */
5      public InvalidRadiusException(double radius) {
6          super("Invalid radius " + radius);
7          this.radius = radius;
8      }
9
10     /** Return the radius */
11     public double getRadius() {
12         return radius;
13     }
14 }
```


References

- ▶ Liang, Chapter 12: Exception Handling and Text I/O