Week11

Programming Fundamentals II

Course Outline

8. GUI	เสนอหัวข้อ Project
9. Event	Lab7
10. Graphic	Lab8
11. Exception	Lab9
12. Testing and debugging	Lab10
13. JAVA Project	เสนอความคืบหน้า Project
14. Generics	-

Final Exam   Lab Exam   Present Project

Week 11 **Exception**

1. Introduction
2. Basic Approach
3. The Exception Class Hierarchy
4. Throwing an Exception
5. Handling Exceptions
6. Finally Block
7. Stack Trace
8. Assertions

Exception

Programming Fundamentals II

Introduction

Exception handling

Exception—an indication of a problem that occurs during a program's execution.

- The name "exception" implies that the problem occurs infrequently.

With exception handling, a program can continue executing (rather than terminating) after dealing with a problem.

- Mission-critical or business-critical computing.
- Robust and fault-tolerant programs (i.e., programs that can deal with problems as they arise and continue executing).

Introduction

Error and Exception

Error

- IO Error
- NoClassDefFoundError
- StackOverflowError

Exception

- AritmaticException
- more ...

Introduction

- `ArrayIndexOutOfBoundsException` occurs when an attempt is made to access an element past either end of an array.
- `ClassCastException` occurs when an attempt is made to cast an object that does not have an *is-a* relationship with the type specified in the cast operator.
- A `NullPointerException` occurs when a `null` reference is used where an object is expected.
- Only classes that extend `Throwable` (package `java.lang`) directly or indirectly can be used with exception handling.

Motivation

Lots of error checking in code makes the code harder to understand

- more complex
- more likely that the code will have errors!

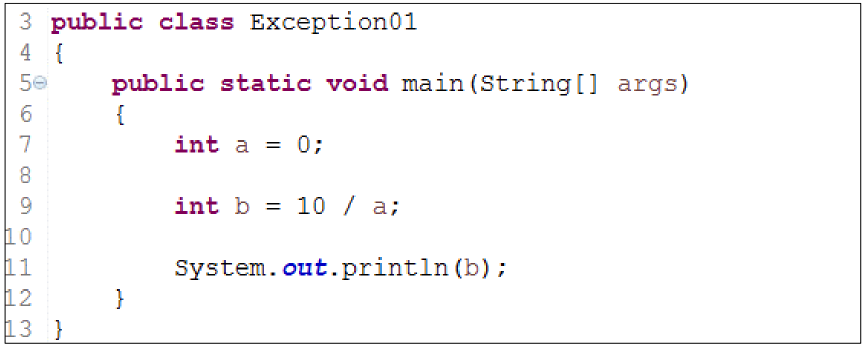
Add error checking to the following C code:

```
int a[SIZE];
y = ...
x = (1.0/a[y]) + (2.0/a[y+1]) + (3.0/a[y+2]);
```

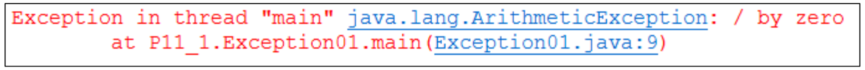
Some Error Checking (in C)!

```
    :
if (y >= SIZE)
  printf(“Array index %d too big\n”, y);
else if (a[y] == 0)
  printf(“First denominator is 0\n”);

if (y+1 >= SIZE)
  printf(“Array index %d too big\n”,  y+1);
else if (a[y+1]== 0)
  printf(“Second denominator is 0\n”);

if (y+2 >= SIZE)
  printf(“Array index %d too big\n”,  y+2);
else if (a[y+2]== 0)
  printf(“Third denominator is 0\n”);
    :
```

Some Error in JAVA

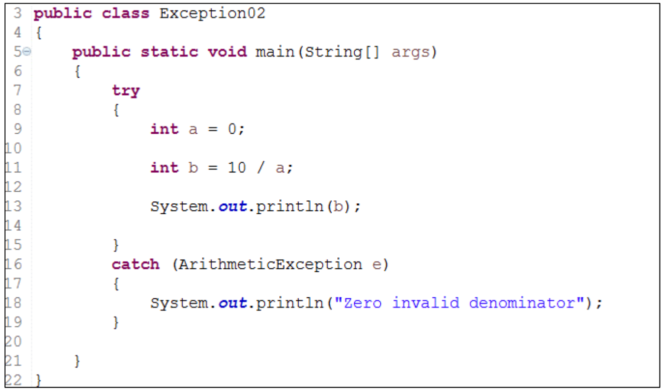


```
public class Exception01
{
    public static void main(String[] args)
    {
        int a = 0;

        int b = 10 / a;

        System.out.println(b);
    }
}
```

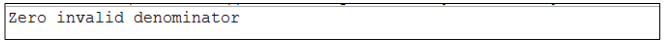


```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at P11_1.Exception01.main(Exception01.java:9)
```

Some Error in JAVA



```
public class Exception02
{
    public static void main(String[] args)
    {
        try
        {
            int a = 0;

            int b = 10 / a;

            System.out.println(b);

        }
        catch (ArithmeticException e)
        {
            System.out.println("Zero invalid denominator");
        }

    }
}
```



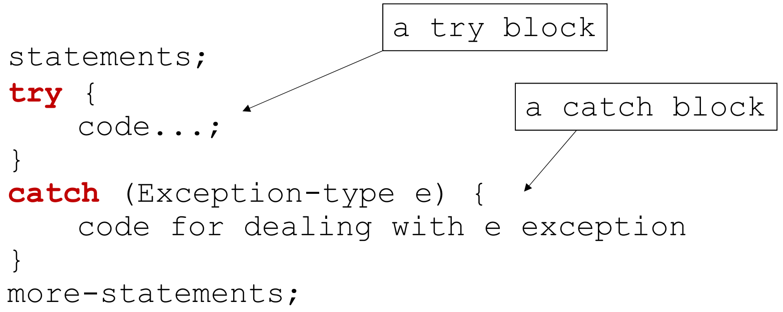
```
Zero invalid denominator
```

Try, Catch

Programming Fundamentals II

Exception Handling (in outline)

Format of code:



```
statements;
try {
    code...;
}
catch (Exception-type e) {
    code for dealing with e exception
}
more-statements;
```

a try block

a catch block

Basic Approach (1/3)

The programmer wraps the error-prone code inside a try block.

If an exception occurs anywhere in the code inside the try block, the catch block is executed immediately

- the block can use information stored in the e object

Basic Approach (2/3)

After the `catch` block (the catch ***handler***) has finished, execution continues after the `catch` block (in `more-statements`).

- execution does **not** return to the `try` block

Basic Approach (3/3)

If the `try` block finishes successfully without causing an exception, then execution skips to the code after the `catch` block

i.e. to `more-statements`

Example: Divide by Zero without Exception Handling

```
// Fig. 11.2: DivideByZeroNoExceptionHandling.java
// Integer division without exception handling.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient(int numerator, int denominator)
   {
      return numerator / denominator; // possible division by zero
   }

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

Fig. 11.2 | Integer division without exception handling. (Part 1 of 3.)

```

      System.out.print("Please enter an integer numerator: ");
      int numerator = scanner.nextInt();
      System.out.print("Please enter an integer denominator: ");
      int denominator = scanner.nextInt();

      int result = quotient(numerator, denominator);
      System.out.printf(
         "%nResult: %d / %d = %d%n", numerator, denominator, result);
   }
} // end class DivideByZeroNoExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 11.2 | Integer division without exception handling. (Part 2 of 3.)

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at DivideByZeroNoExceptionHandling.quotient(
           DivideByZeroNoExceptionHandling.java:10)
        at DivideByZeroNoExceptionHandling.main(
           DivideByZeroNoExceptionHandling.java:22)
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at DivideByZeroNoExceptionHandling.main(
            DivideByZeroNoExceptionHandling.java:20)
```

Fig. 11.2 | Integer division without exception handling. (Part 3 of 3.)

Catching Math Errors

```
int x = 0;
int y;
    :
try
{
     y = 1/x;
      :
}
catch (ArithmeticException e)
{
     System.out.println(e);
     ...;
     y = 0;
}

System.out.println(“y is ” + y);
```

any Java code is allowed here

Many Catch Blocks

There can be many catch blocks associated with a try block

- the choice of which to use is based on matching the exception object (e) with the argument type of each catch block

- after a catch handler has finished, execution continues after all the handlers

Code Format

```
statements;
try {
  code...;
}
catch (NullPointerException e) {
  code for dealing with a NULL pointer
exception
}
catch (IOException e) {
  code for dealing with an IO exception
}
catch (MyOwnException e) {
  code for dealing with a user-defined
exception
}
more-statements;
```

Example: Handling ArithmeticExceptions and InputMismatchExceptions

- Prior examples that input numeric values assumed that the user would input a proper integer value.

- Users sometimes make mistakes and input noninteger values.

- An InputMismatchException occurs when Scanner method nextInt receives a String that does not represent a valid integer.

Example: Handling ArithmeticExceptions and InputMismatchExceptions

- The application in Example exception handling to

  process any

`ArithmeticExceptions`

`and`

`InputMistmatchExceptions.`

Example: Handling ArithmeticExceptions and InputMismatchExceptions

try block encloses

- code that might throw an exception
- code that should not execute if an exception occurs.

Consists of the keyword try followed by a block of code enclosed in curly braces.

Example: Handling ArithmeticExceptions and InputMismatchExceptions

```java
// Fig. 11.3: DivideByZeroWithExceptionHandling.java
// Handling ArithmeticExceptions and InputMismatchExceptions.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient(int numerator, int denominator)
      throws ArithmeticException
   {
      return numerator / denominator; // possible division by zero
   }

   public static void main(String[] args)
   {
      Scanner scanner = new Scanner(System.in);
      boolean continueLoop = true; // determines if more input is needed

```

Fig. 11.3 | Handling ArithmeticExceptions and InputMismatchExceptions. (Part 1 of 4.)

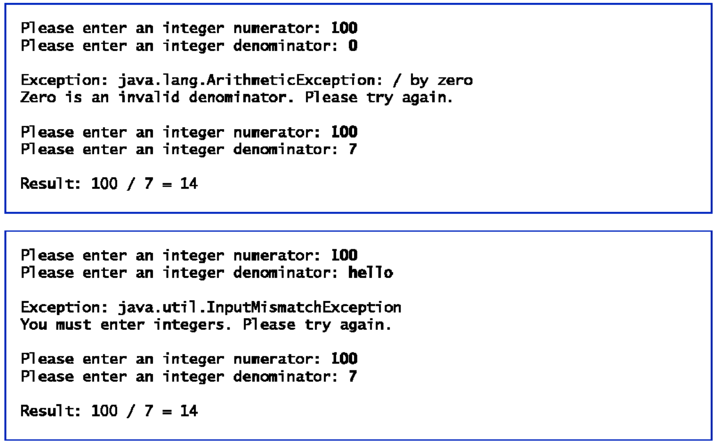
```java
      do
      {
         try // read two numbers and calculate quotient
         {
            System.out.print("Please enter an integer numerator: ");
            int numerator = scanner.nextInt();
            System.out.print("Please enter an integer denominator: ");
            int denominator = scanner.nextInt();

            int result = quotient(numerator, denominator);
            System.out.printf("%nResult: %d / %d = %d%n", numerator,
               denominator, result);
            continueLoop = false; // input successful; end looping
         }
         catch (InputMismatchException inputMismatchException)
         {
            System.err.printf("%nException: %s%n",
               inputMismatchException);
            scanner.nextLine(); // discard input so user can try again
            System.out.printf(
               "You must enter integers. Please try again.%n%n");
         }
```

Fig. 11.3 | Handling ArithmeticExceptions and InputMismatchExceptions. (Part 2 of 4.)

```java
         catch (ArithmeticException arithmeticException)
         {
            System.err.printf("%nException: %s%n", arithmeticException);
            System.out.printf(
               "Zero is an invalid denominator. Please try again.%n%n");
         }
      } while (continueLoop);
   }
} // end class DivideByZeroWithExceptionHandling
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Fig. 11.3 | Handling ArithmeticExceptions and InputMismatchExceptions. (Part 3 of 4.)

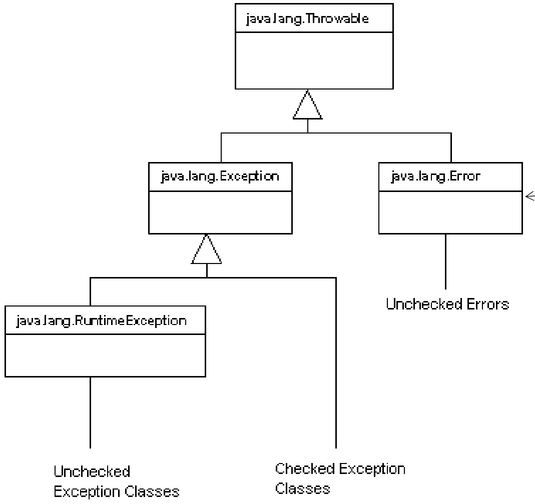


```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

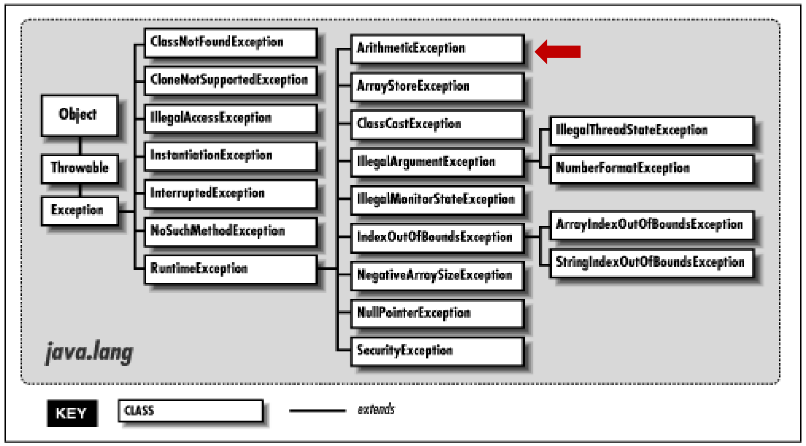
Fig. 11.3 | Handling `ArithmeticException`s and `InputMismatchException`s. (Part 4 of 4.)

4. The Exception Class Hierarchy


java.lang.Throwable
java.lang.Exception
java.lang.Error
Unchecked Errors
java.lang.RuntimeException
Unchecked Exception Classes
Checked Exception Classes

for errors that occur inside the JVM, not in your code

In More Detail



1 Two Exception Categories

1.1 Checked exceptions

- subclasses of Exception
- recovery should be possible for these types of errors
- your code must include try-catch blocks for these or the compiler will reject your program

  e.g. IOException

Text IO (Checked exceptions)

IO can generate lots of exceptions, but usually the program can recover

- e.g. file not found, so look somewhere else

Most IO methods can produce `java.io.IOException`

- a checked exception which your code **must** handle with try-catch blocks

Text Input From File

Use the `FileReader` class.

Use `BufferedReader` for line-based input:

- open a file
- read from the file
- close the file

Failure at any point results in an `IOException.`

Text Input From File

```
try {
    BufferedReader reader =
        new BufferedReader(new
FileReader("filename"));
    String line = reader.readLine();
    while(line != null) {
        do something with line
        line = reader.readLine();
  }
    reader.close();
}
catch(FileNotFoundException e) {
    // the specified file could not be found
}
catch(IOException e) {
    // something went wrong with reading or closing
}
```

Two Exception Categories

1.2 Unchecked exceptions

- subclasses of RuntimeException
- exceptions of this type usually mean that your program should terminate
- the compiler does not force you to include try-catch blocks for these kinds of exceptions e.g. ArithmeticException

Checking Maths (Unchecked exceptions)

```
int x = 0;
int y;
 :
try {
  y = 1/x;
   :
}
catch(ArithmeticException e)
{  ...;
   y = 0;
}

System.out.println(“y is “ + y);
```

Checking Maths (Unchecked exceptions)

```
int x = 0;
int y;
  :
y = 1/x;
     :

System.out.println(“y is “ + y);
```

```
a try-catch block does
not need to be included
```

Throw

Programming Fundamentals II

Throwing an Exception

Exceptions are caused (thrown or raised) by the JVM.

Also, the programmer can throw an exception by using:

```
throw e
```

Example

```
private double safeSqrt(double x)
{
  try {
    if (x < 0.0)
   throw new ArithmeticException();
    . . .;
  }
  catch (ArithmeticException e)
  {  x =  0.0; }
   . . . ;
  return sqrt(x);
}
```

Handling Exceptions

Exceptions thrown by a method can be either:

- caught by the method's catch handler(s)

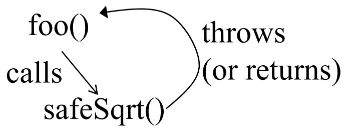
  - we've seen examples already

- or be listed in the method's `throws` declaration

Example

```
double safeSqrt(double x) throws
                    ArithmeticException
{
  if (x < 0.0)
      throw new ArithmeticException();

  . . .;
  return sqrt(x);
}
```

Example



```
void foo(double x)
{
     double result;
     try {
          result = safeSqrt(x);
     }
     catch(ArithmeticException e) {
          System.out.println(e);
          result = -1;
     }

     System.out.println("result: " + result);
}
```

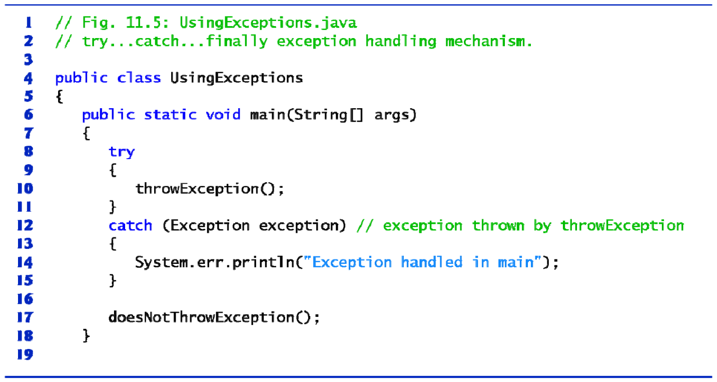
Finally

Programming Fundamentals II

Finally Block

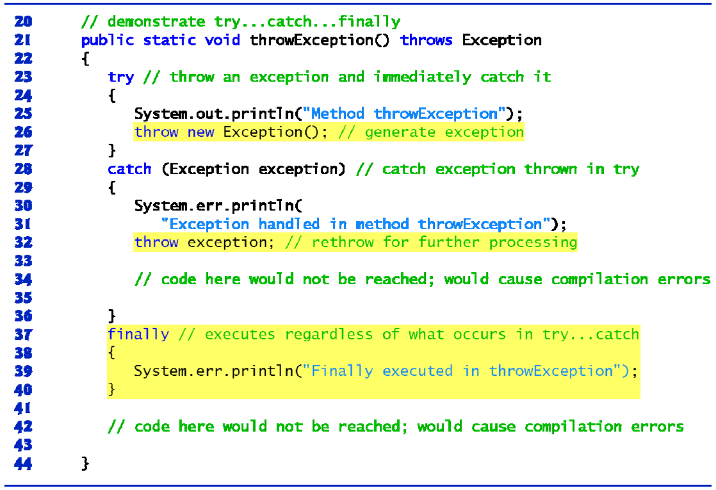
- `finally` block will execute whether or not an exception is thrown in the corresponding `try` block.
- `finally` block will execute if a `try` block exits by using a `return`, `break` or `continue` statement or simply by reaching its closing right brace.
- `finally` block will not execute if the application exits early from a `try` block by calling method *System.exit.*

Example



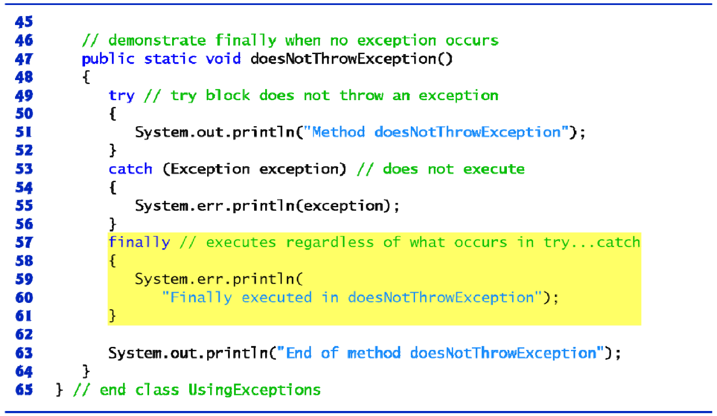
```java
// Fig. 11.5: UsingExceptions.java
// try...catch...finally exception handling mechanism.

public class UsingExceptions
{
   public static void main(String[] args)
   {
      try
      {
         throwException();
      }
      catch (Exception exception) // exception thrown by throwException
      {
         System.err.println("Exception handled in main");
      }

      doesNotThrowException();
   }

```

Fig. 11.5 | try...catch...finally exception-handling mechanism. (Part 1 of 4.)



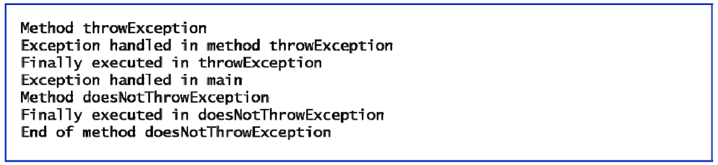
```java
   // demonstrate try...catch...finally
   public static void throwException() throws Exception
   {
      try // throw an exception and immediately catch it
      {
         System.out.println("Method throwException");
         throw new Exception(); // generate exception
      }
      catch (Exception exception) // catch exception thrown in try
      {
         System.err.println(
            "Exception handled in method throwException");
         throw exception; // rethrow for further processing

         // code here would not be reached; would cause compilation errors

      }
      finally // executes regardless of what occurs in try...catch
      {
         System.err.println("Finally executed in throwException");
      }

      // code here would not be reached; would cause compilation errors

   }
```

Fig. 11.5 | try...catch...finally exception-handling mechanism. (Part 2 of 4.)



```

   // demonstrate finally when no exception occurs
   public static void doesNotThrowException()
   {
      try // try block does not throw an exception
      {
         System.out.println("Method doesNotThrowException");
      }
      catch (Exception exception) // does not execute
      {
         System.err.println(exception);
      }
      finally // executes regardless of what occurs in try...catch
      {
         System.err.println(
            "Finally executed in doesNotThrowException");
      }

      System.out.println("End of method doesNotThrowException");
   }
} // end class UsingExceptions
```

Fig. 11.5 | try...catch...finally exception-handling mechanism. (Part 3 of 4.)



```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException
```

Fig. 11.5 | try...catch...finally exception-handling mechanism. (Part 4 of 4.)

Stack Trace

- If no handler is called, then the system prints a **stack trace** as the program terminates
  - it is a list of the called methods that are waiting to return when the exception occurred
  - **very useful** for debugging/testing
- The stack trace can be printed by calling **printStackTrace()**

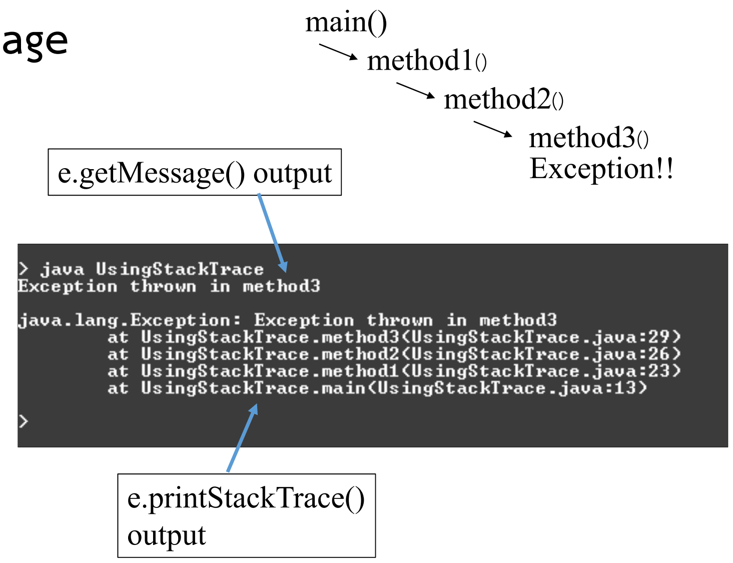
Using a Stack Trace

```
// The getMessage and printStackTrace methods
public class UsingStackTrace
{
      public static void main( String args[] )
      {
       try{
          method1();
        }
       catch ( Exception e) {
          System.err.println(e.getMessage() + "\n");
          e.printStackTrace();
        }
      }
 }// end of main()
```

Using a Stack Trace

```
 public static void method1() throws Exception
   { method2(); }

   public static void method2() throws Exception
   { method3(); }

   public static void method3() throws Exception
   {
      throw new Exception(
                        "Exception thrown in method3" );
   }

} // end of UsingStackTrace class
```

Usage



```
> java UsingStackTrace
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingStackTrace.method3(UsingStackTrace.java:29)
        at UsingStackTrace.method2(UsingStackTrace.java:26)
        at UsingStackTrace.method1(UsingStackTrace.java:23)
        at UsingStackTrace.main(UsingStackTrace.java:13)

>
```

Stack Unwinding and Obtaining Information from an Exception Object

- `Stack unwinding`—When an exception is thrown but not caught in a particular scope, the method-call stack is “unwound”
- An attempt is made to `catch` the exception in the next outer `try` block.
- All local variables in the unwound method go out of scope and control returns to the statement that originally invoked that method.
- If a `try` block encloses that statement, an attempt is made to `catch` the exception.
- If a `try` block does not enclose that statement or if the exception is not caught, stack unwinding occurs again.

Stack Unwinding and Obtaining Information from an Exception Object

- exception.getMessage ()
- exception.printStackTrace()
- exception.getStackTrace()

Example

```
// Fig. 11.6: UsingExceptions.java
// Stack unwinding and obtaining data from an exception object.

public class UsingExceptions
{
   public static void main(String[] args)
   {
      try
      {
         method1();
      }
      catch (Exception exception) // catch exception thrown in method1
      {
         System.err.printf("%s%n%n", exception.getMessage());
         exception.printStackTrace();

         // obtain the stack-trace information
         StackTraceElement[] traceElements = exception.getStackTrace();

         System.out.printf("%nStack trace from getStackTrace:%n");
         System.out.println("Class\t\tFile\t\t\tLine\tMethod");

```

Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 1 of 4.)

```
         // loop through traceElements to get exception description
         for (StackTraceElement element : traceElements)
         {
            System.out.printf("%s\t", element.getClassName());
            System.out.printf("%s\t", element.getFileName());
            System.out.printf("%s\t", element.getLineNumber());
            System.out.printf("%s%n", element.getMethodName());
         }
      }
   } // end main

   // call method2; throw exceptions back to main
   public static void method1() throws Exception
   {
      method2();
   }

   // call method3; throw exceptions back to method1
   public static void method2() throws Exception
   {
      method3();
   }
```

Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 2 of 4.)

```

   // throw Exception back to method2
   public static void method3() throws Exception
   {
      throw new Exception("Exception thrown in method3");
   }
} // end class UsingExceptions
```

Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 3 of 4.)

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:49)
        at UsingExceptions.method2(UsingExceptions.java:43)
        at UsingExceptions.method1(UsingExceptions.java:37)
        at UsingExceptions.main(UsingExceptions.java:10)

Stack trace from getStackTrace:
Class           File                    Line    Method
UsingExceptions UsingExceptions.java    49      method3
UsingExceptions UsingExceptions.java    43      method2
UsingExceptions UsingExceptions.java    37      method1
UsingExceptions UsingExceptions.java    10      main
```

Fig. 11.6 | Stack unwinding and obtaining data from an exception object. (Part 4 of 4.)

Assertions

- When implementing and debugging a class, it's sometimes useful to state conditions that should be true at a particular point in a method.
- Assertions help ensure a program's validity by catching potential bugs and identifying possible logic errors during development.
- Preconditions and postconditions are two types of assertions.

Assertions

- Java includes two versions of the assert statement for validating assertions programatically.
- `assert` evaluates a `boolean` expression and, if `false`, throws an AssertionError (a subclass of `Error`).
  - assert *`expression`*;
    - throws an `AssertionError` if *expression is* *`false`*.
  - assert *`expression1`* : *`expression2`*;
    - evaluates *expression1* and throws an `AssertionError` with *expression2* as the error message if expression1 is `false`.
- Can be used to programmatically implement preconditions and postconditions or to verify any other *intermediate* states that help you ensure your code is working correctly.

Example

```
// Fig. 11.8: AssertTest.java
// Checking with assert that a value is within range
import java.util.Scanner;

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

      System.out.print("Enter a number between 0 and 10: ");
      int number = input.nextInt();

      // assert that the value is >= 0 and <= 10
      assert (number >= 0 && number <= 10) : "bad number: " + number;

      System.out.printf("You entered %d%n", number);
   }
} // end class AssertTest
```

```
Enter a number between 0 and 10: 5
You entered 5
```

Fig. 11.8 | Checking with `assert` that a value is within range. (Part 1 of 2.)

```
Enter a number between 0 and 10: 50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:15)
```

Fig. 11.8 | Checking with `assert` that a value is within range. (Part 2 of 2.)

END