Faculty of Computers and Information
Menoufia University

Computer Language-1

Lecture 3

Exception Handling

Dr. Eman Meslhy

2020-2021

Handling Exceptions

- An exception is an object that is generated as the result of an error or an unexpected event.
- Exception are said to have been “thrown.”
- It is the programmers responsibility to write code that detects and handles exceptions.
- Unhandled exceptions will crash a program.
- Java allows you to create exception handlers.

```
 int x = 10 , y = 0 ;

System.out.println (x/y);
```

Divide by Zero

Exception

- An exception is an error or a condition that prevents execution of a program from proceeding normally.
- For example:

```
int[] x = {1, 5, 7};
System.out.println(x[1]);
System.out.println(x[3]);
System.out.println(x[0]);
System.out.println(x[2]);
```

ArrayIndexOutOfBoundsException

```
int x = 5;
int y = 0;
int z = x/y;
System.out.println(z);
```

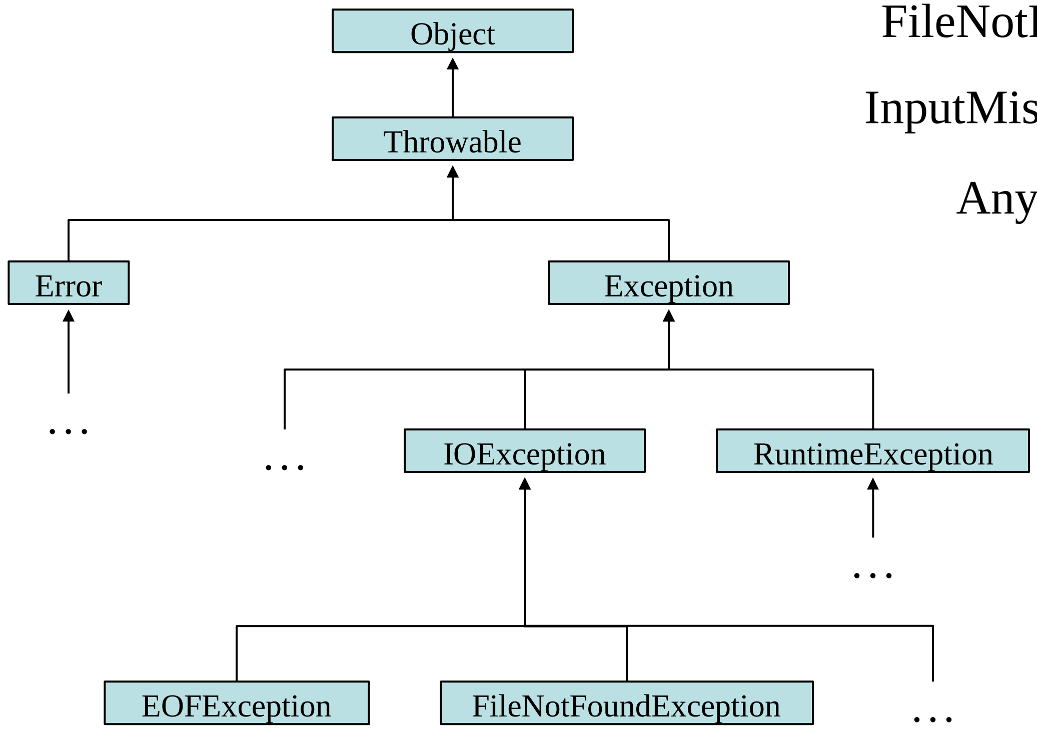
ArithmeticException

- If the exception is not handled, the program will terminate abnormally.

Exception Classes

- An *exception handler* is a section of code that gracefully responds to exceptions.
- An exception is an object.
- Exception objects are created from classes in the Java API hierarchy of exception classes.
- All of the exception classes in the hierarchy are derived from the `Throwable` class.
- `Error` and `Exception` are derived from the `Throwable` class.

Exception Classes



FileNotFoundException is Exception

InputMisMatchException is Exception

AnyException is Exception

Handling Exceptions

- To handle an exception, you use a *try* statement.

- **`try`**
- **`{`**
  - ***(try block statements...)***
- **`}`**
- **`catch (`*`ExceptionType ParameterName`*`)`**
- **`{`**
  - ***(catch block statements...)***
- **`}`**
- First the keyword `try` indicates a block of code will be attempted.

Handling Exceptions

- After the try block, a `catch` clause appears.
- A catch clause begins with the key word `catch`:
- **`catch (`*`ExceptionType ParameterName`*`)`**
  - *`ExceptionType`* is the name of an exception class and
  - *`ParameterName`* is a variable name which will reference the exception object if the code in the try block throws an exception.
- The code that immediately follows the catch clause is known as a *catch block* .
- The code in the catch block is executed if the try block throws an exception.

Exception Handling

- Exception handling enables a program to deal with exceptional situations and continue its normal execution.

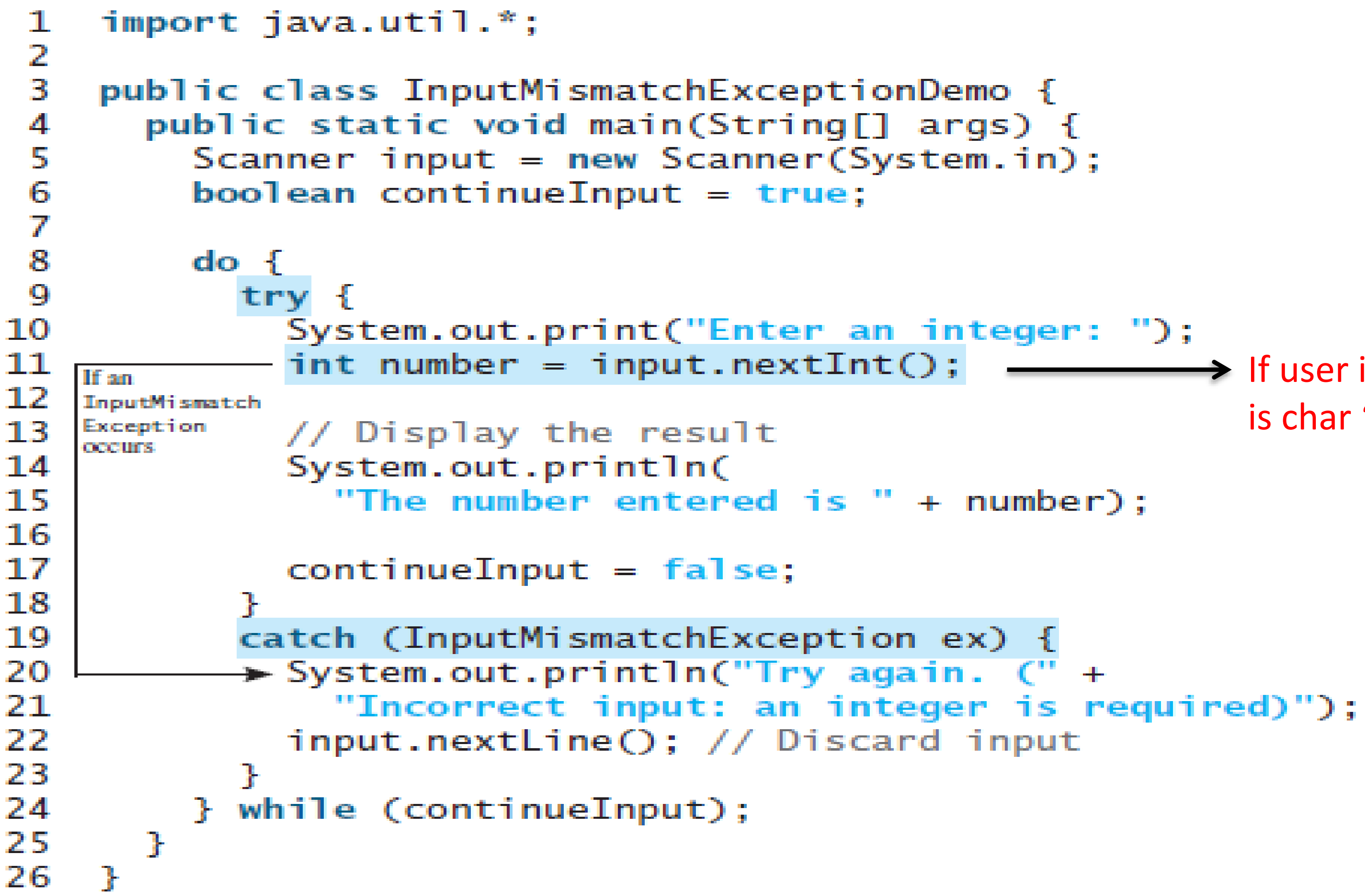
```
Try {
    Statement or method that may throw an exception
 }
Catch (type ex){
   Code to process the exception
 }
```

```
int[] x = {1, 5, 7}
Try {
    System.out.println(x[3]);
} Catch (ArrayIndexOutOfBoundsException ex){
   System.out.println(“You access an index out of bound”)
 }
System.out.println(x[0]);
System.out.println(x[2]);
```

Handling Exceptions

- This code is designed to handle a `FileNotFoundException` if it is thrown.
- **`try`**
- **`{`**
  - **`File file = new File ("MyFile.txt"); Scanner inputFile = new Scanner(file);`**
- **`}`**
- **`catch (FileNotFoundException e)`**
- **`{`**
  - **`System.out.println("File not found.");`**
- **`}`**
- The Java Virtual Machine searches for a `catch` clause that can deal with the exception.

Exception handling Example


If an
InputMismatch
Exception
occurs

```java
import java.util.*;

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

    do {
      try {
        System.out.print("Enter an integer: ");
        int number = input.nextInt();

        // Display the result
        System.out.println(
          "The number entered is " + number);

        continueInput = false;
      }
      catch (InputMismatchException ex) {
        System.out.println("Try again. (" +
          "Incorrect input: an integer is required)");
        input.nextLine(); // Discard input
      }
    } while (continueInput);
  }
}
```

If user input is char ‘a’?

Quiz-1

```
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) {
                  System.out.println(“Error”);
               }
           }
      }
}
```

What is the output?

0   Error
1   Error

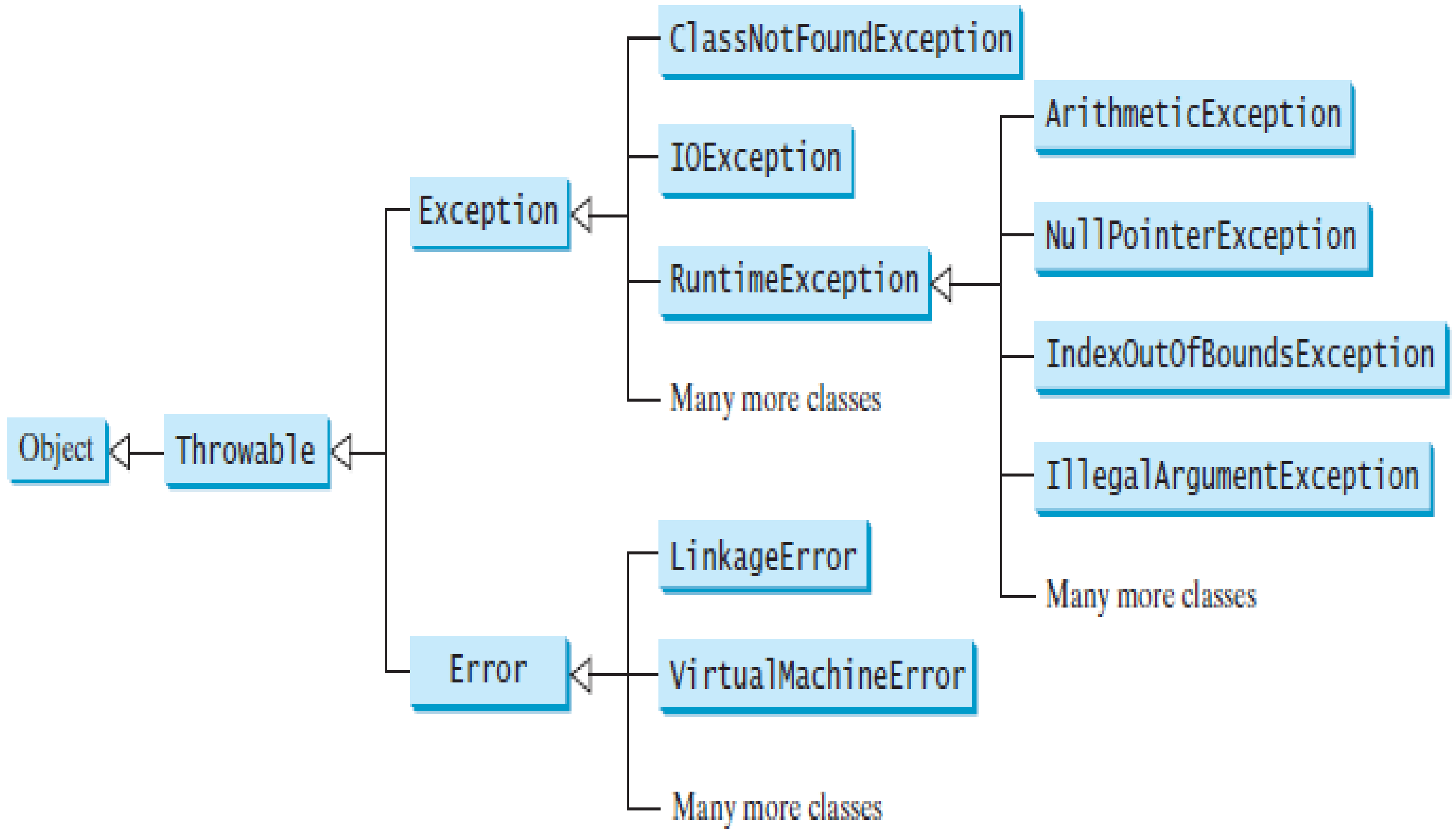
Quiz-2

```
public class Main {
    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) {
          System.out.println(“Error”);
           }
     }
}
```

What is the output?

0 Error

Exception Types



Quiz-3

```
public class Test {
  public static void main(String[] args) {
```
ArithmeticException
```
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
```
ArrayIndexOutOfBoundsException
```
  }
}
```

(b)

```
public class Test {
  public static void main(String[] args) {
```
StringIndexOutOfBoundsException
```
  }
}
```

(c)

```
public class Test {
  public static void main(String[] args) {
    Objec
    Strin
```
ClassCastException
```
  }
}
```

(d)

```
public class Test {
  public static void main(String[] args) {
    Obj
    Sys                                      );
```
NullPointerException
```
  }
}
```

(e)

```
public class
  public stat                   ng[] args) {
    System.ou                   );
  }
}
```

?

(f)

Quiz 4

```
public class Main {
  public static void main(String[] args) {

System.out.println(1.0 / 0);

          }
          catch (Exception ex) {
         System.out.println("Error");
          }
     }
}
```

Infinity

Infinity

- 1/0 is a division of two *int*s, and throws an exception because you can't divide by integer zero.
- However, 0.0 is a literal of type *double*, and Java will use a floating-point division.
- The IEEE floating-point specification has special values for dividing by zero (among other thing), one of these is double.

Quiz 5

```
public class Main {
    public static void main(String[] args) {

System.out.println(1 / 0.0);

            }
            catch (Exception ex) {
           System.out.println(“Error”);
            }
     }
}
```

Infinity

Handling Multiple Exceptions

- The code in the try block may be capable of throwing more than one type of exception.
- A `catch` clause needs to be written for each type of exception that could potentially be thrown.
- The JVM will run the first compatible `catch` clause found.
- The `catch` clauses must be listed from most specific to most general.

Exception Handlers

- There can be many polymorphic catch clauses.
- A try statement may have only one `catch` clause for each specific type of exception.

```
try
{
    number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
    System.out.println("Bad number format.");
}
catch (NumberFormatException e) // ERROR!!!
{
    System.out.println(str + " is not a number.");
}
```

Exception Handlers

- The `NumberFormatException` class is derived from the
- `IllegalArgumentException` class.
- **`try`**
- **`{`**
  - **`number = Integer.parseInt(str);`**
- **`}`**
- **`catch (IllegalArgumentException e)`**
- **`{`**
  - **`System.out.println("Bad number format.");`**
- **`}`**
- **`catch (NumberFormatException e)`** **`// ERROR!!!`**
- **`{`**
  - **`System.out.println(str + " is not a number.");`**
- **`}`**

```
Main.java:24: error: exception NumberFormatException has already been caught
catch(NumberFormatException e2)
```

Exception Handlers

- The previous code could be rewritten to work, as follows, with no errors:

```
try
{
   number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
   System.out.println(str + " is not a number.");
}
catch (IllegalArgumentException e) //OK
{
   System.out.println("Bad number format.");
}
```

Quiz 6

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

(a)

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

(b)

b

The `finally` Clause

- The try statement may have an optional `finally` clause.
- If present, the `finally` clause must appear after all of the `catch` clauses.
- **`try`**
- **`{`**
  - ***(try block statements...)***
- **`}`**
- **`catch (`*`ExceptionType ParameterName`*`)`**
- **`{`**
  - ***(catch block statements...)***
- **`}`**
- **`finally`**
- **`{`**
  - ***(finally block statements...)***
- **`}`**

The `finally` Clause

- The *finally block* is one or more statements,
  - that are always executed after the try block has executed and
  - after any catch blocks have executed if an exception was thrown.
- The statements in the finally block execute whether an exception occurs or not.

Throwing Exceptions

- You can write code that:
  - throws one of the standard Java exceptions, or
  - an instance of a custom exception class that you have designed.
- The `throw` statement is used to manually throw an exception.
- **`throw new`** ***`ExceptionType(MessageString);`***
- The `throw` statement causes an exception object to be created and thrown.

Throwing Exceptions

- The *MessageString* argument contains a custom error message that can be retrieved from the exception object’s `getMessage` method.
- If you do not pass a message to the constructor, the exception will have a null message.
- **`throw new Exception("Out of fuel");`**
- ***Example:***
- if (Length ==Width)
- {
- throw new IllegalArgumentException(“In Rectangle ,The Length must be different from width.");
- }

```
try{
  int x , y=10;
  Scanner s= new Scanner(System.in);
    x= s.nextInt ( );
    if (x==0)
   throw new IllegalArgumentException("Must be more than 0");
   System.out.println(y/x); }
catch(ArithmeticException e){
System.out.println("Error");}
  catch(IllegalArgumentException e2)
  {  System.out.println("wrong value");}
catch (InputMismatchException e3){
System.out.println("Enter only numeric value"); }
System.out.println("Final");
}
```

```
0
wrong value
Final
```

```
try{
  int x , y=10;
  Scanner s= new Scanner(System.in);
    x= s.nextInt ( );
    if (x==0)
   throw new IllegalArgumentException("Must be more than 0");
   System.out.println(y/x); }
catch(ArithmeticException e){
System.out.println("Error");}
catch (InputMismatchException e3){
System.out.println("Enter only numeric value"); }
System.out.println("Final");
}
```

```
0
Exception in thread "main" java.lang.IllegalArgumentException: Must
be more than 0   at Main.main(Main.java:17)
```

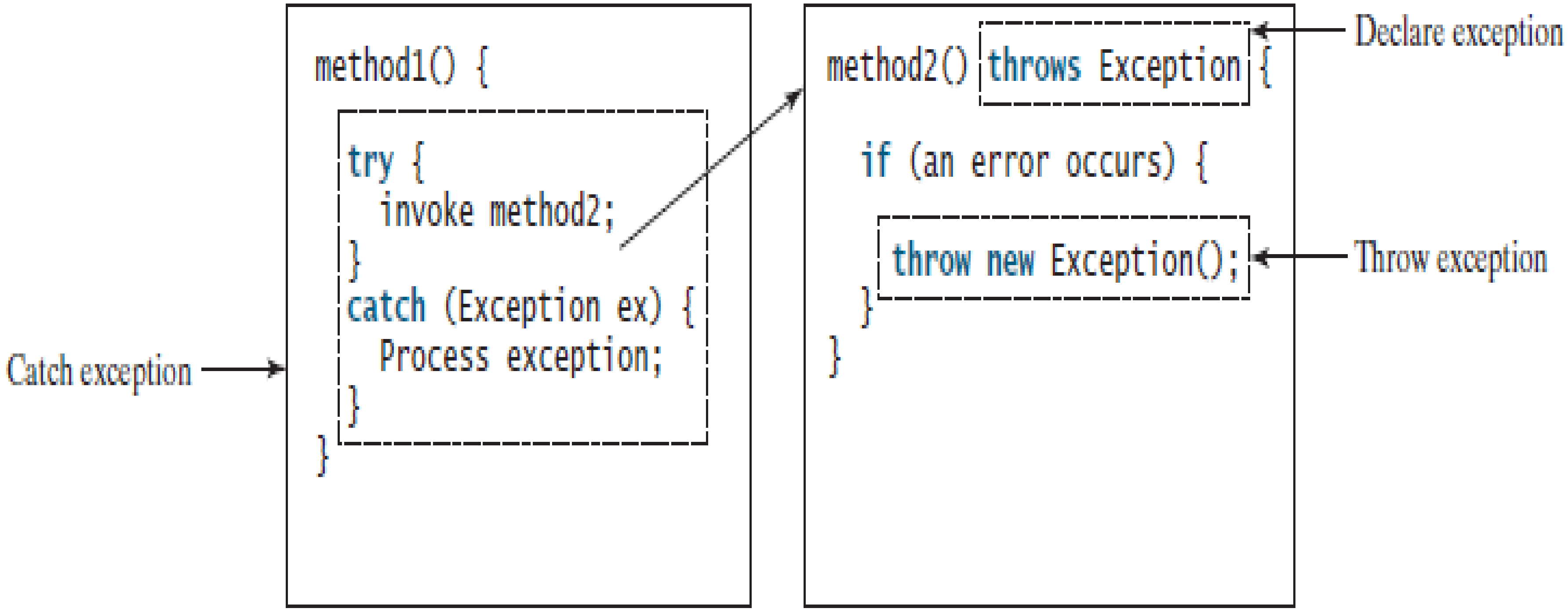
Exception Types

- *RuntimeException* and *Error* are known as unchecked e
- A
- T l

v

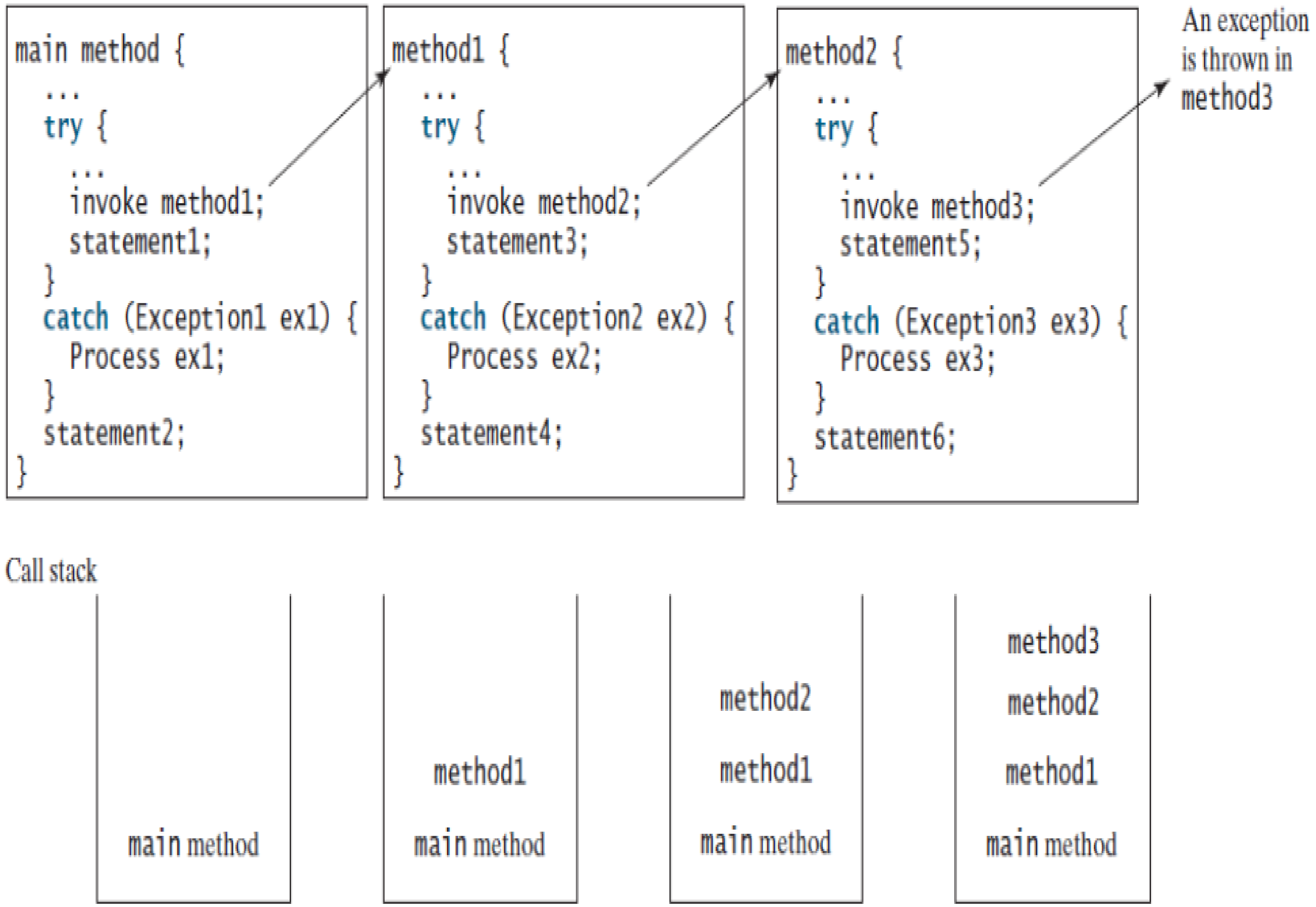
```
public class Test {
    public static void main(String[] args) {
          try {
              FileReader fr = new FileReader("……");  →  FileNotFoundException
           }
           catch (FileNotFoundException ex) {
           }
     }
}
```

More on Exception Handling



```
method1() {
  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {
  if (an error occurs) {
    throw new Exception();
  }
}
```



```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

Declaring Exceptions

- Every method must state the types of checked exceptions it might throw in method header.
- For example:

public void *myMethod1()* ***throws*** *Exception1*

public void *myMethod2()* ***throws*** *Exception1, … , ExceptionN*

```
public int getIndex(int index, int[] arr) {
   int value =  arr[index];
   return value;
}
```

```
public int getIndex(int index, int[] arr) throws ArrayIndexOutOfBoundsException
{
   int value =  arr[index];
   return value;
}
```

Throwing Exceptions

- A program that detects an error can create an instance of an appropriate exception type and throw it.
- For example:

Exception ex = new Exception();

throw ex;

OR

throw new Exception();

```
public int getIndex(int index, int[] arr) throws ArrayIndexOutOfBoundsException
{
    if ( index >= arr.length){
        throw new ArrayIndexOutOfBoundsException("...........");
    } else{
        int value =  arr[index];
        return value;
    }
}
```

Catching Exceptions

- When an exception is thrown, it can be caught and handled in a try-catch block.

```
public static int getIndex(int index, int[] arr) throws ArrayIndexOutOfBoundsException
{
     if ( index >= arr.length){
          throw new ArrayIndexOutOfBoundsException("You access index out of bounds");
      } else{
          int value =  arr[index];
          return value;
    }
}
```

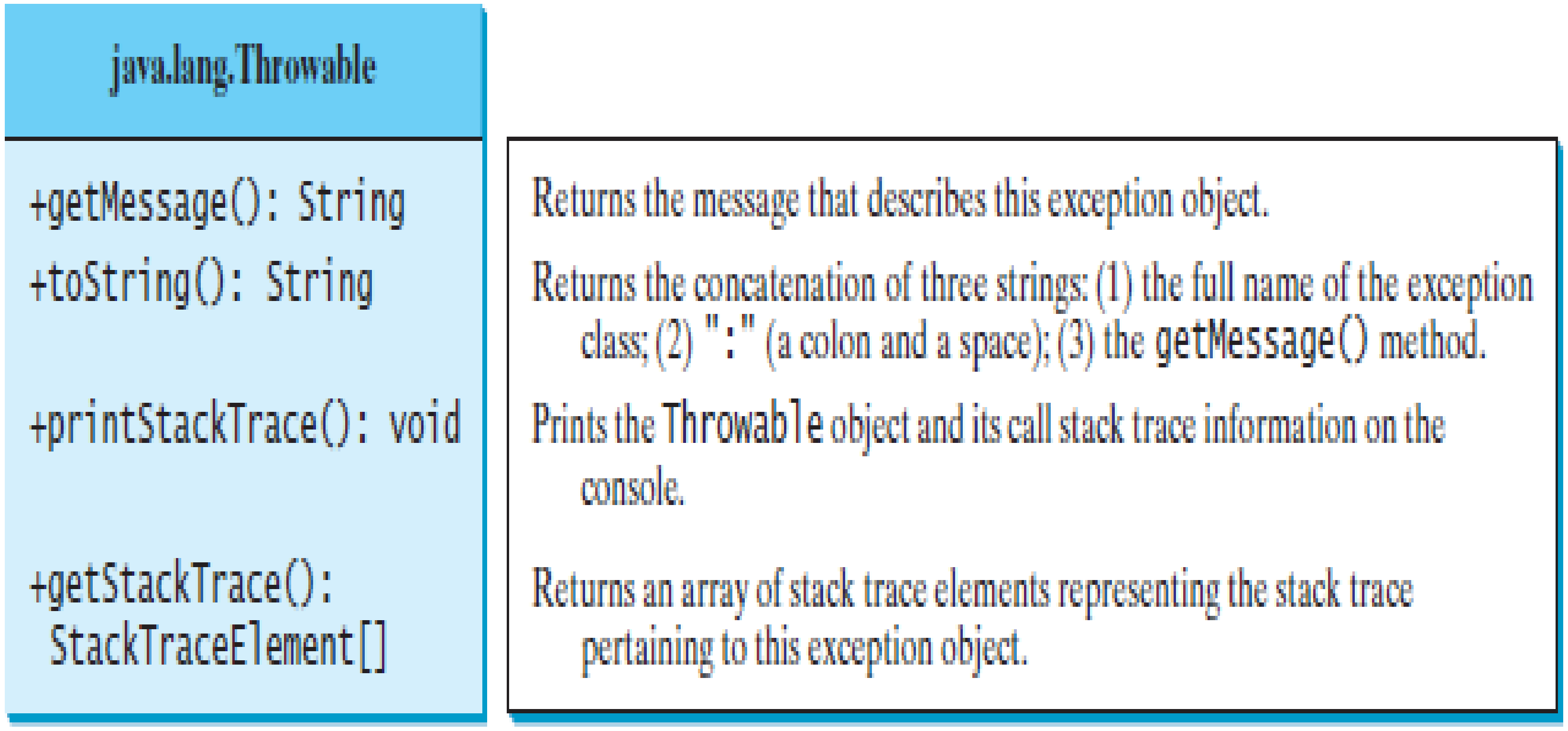
```
int[] arr = {3, 5, 7};
try {
   int x = getIndex(4);
 }
 catch (ArrayIndexOutOfBoundsException  ex) {
     System.out.println(ex.getMessage());
 }
```

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;
}
```

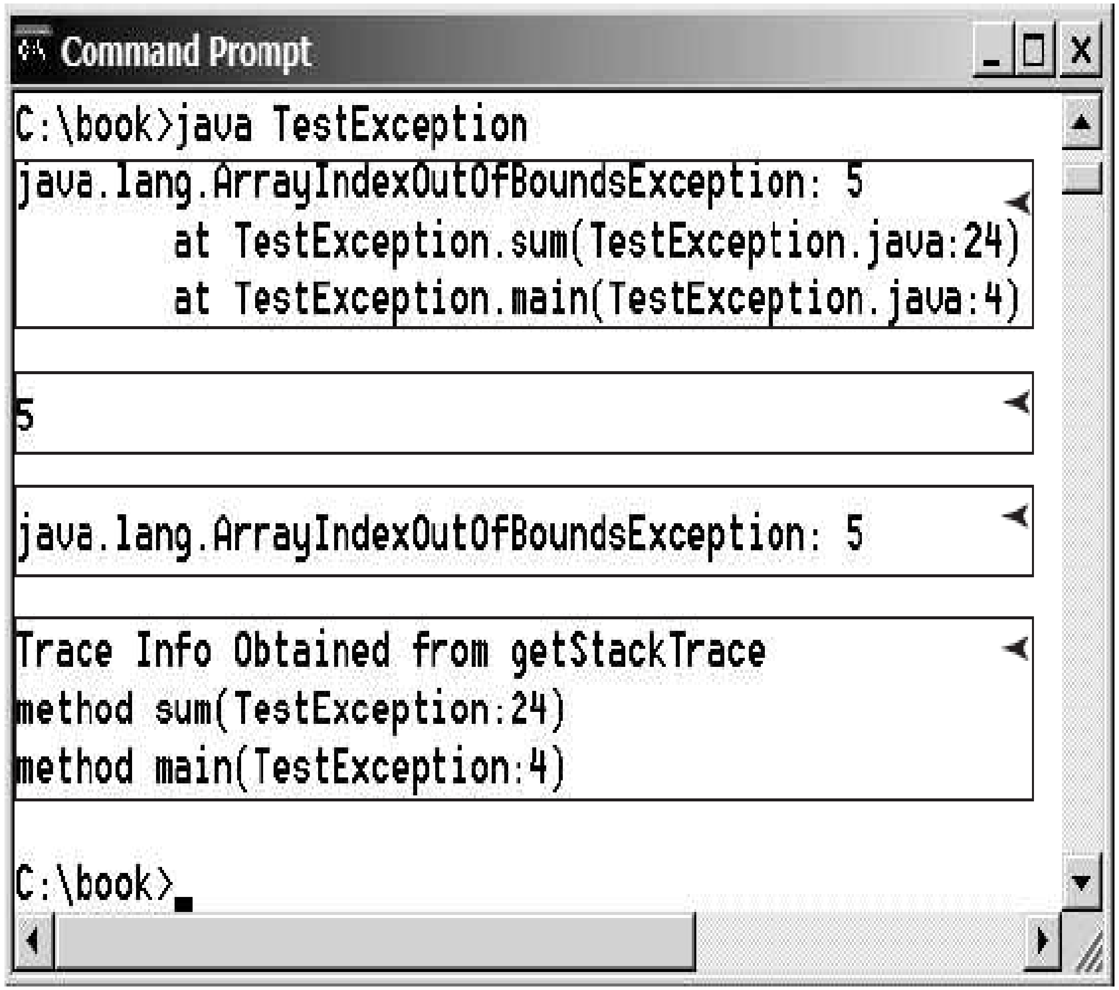
Getting Information from Exceptions

- An exception object contains valuable information about the exception.



java.lang.Throwable	
+getMessage(): String	Returns the message that describes this exception 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 exception object.

Getting Information from Exceptions



```
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()

UsinggetStackTrace()

Finally Clause

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

The **finally** clause is always executed regardless whether an exception occurred or not.

Finally Clause

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

The **finally** clause is always executed regardless whether an exception occurred or not.

Finally Clause

- If no exception arises in the **try** block, **finalStatements** is executed, and the next statement after the **try** statement is executed.
- If a statement causes an exception in the **try** block that is caught in a **catch** block, the rest of the statements in the **try** block are skipped, the **catch** block is executed, and the **finally** clause is executed. The next statement after the **try** statement is executed.
- If one of the statements causes an exception that is not caught in any **catch** block, the other statements in the **try** block are skipped, the **finally** clause is executed, and the exception is passed to the caller of this method.

Finally Clause

```
try {
   statement1;
   statement2;
   statement3;
}
catch (Exception1 ex1) {
   statement4;
}
finally {
   statement5;
}
statement6;
```

If no Exception
statement1
statement2
statement3
statement5
statement6

If caught Exception in statement2
statement1
statement2
statement4
statement5
statement6

If not caught Exception in statement2
statement1
statement2
statement5

Quiz 6

Suppose that statement2 causes an exception in the following statement:

```
try {
statement1;
statement2;
statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
throw ex2;
}
finally {
statement4;
}
statement5;
```

- If no exception occurs, will **statement4** be executed, and will **statement5** be executed?
- If the exception is of type **Exception1**, will **statement4** be executed, and will **statement5** be executed?
- If the exception is of type **Exception2**, will **statement4** be executed, and will **statement5** be executed?
- If the exception is not **Exception1** nor **Exception2**, will **statement4** be executed, and will **statement5** be executed?

Defining Custom Exception Classes

- You can define a custom exception class by extending the **java.lang.Exception** class. Java provides quite a few exception classes. Use them whenever possible instead of defining your own exception classes. However, if you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from **Exception** or from a subclass of **Exception**, such as **IOException**.

Defining Custom Exception Classes

```
public class NonNegativeException extends Exception{
    private int value ;

    public NonNegativeException(int value){
        super("Negative value: " + value);
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

```
public void setAge(int age) throws NonNegativeException{
    if (age <= 0) {
        throw new NonNegativeException(age);
    } else {
        this.age = age;
    }
}
```

ArrayList Class

The `ArrayList` Class

- Similar to an array, an `ArrayList` allows object storage
- Unlike an array, an `ArrayList` object:
  - Automatically expands when a new item is added
  - Automatically shrinks when items are removed
- Requires:
- `import java.util.ArrayList;`

Creating an `ArrayList`

- `ArrayList<String> nameList = new ArrayList<String>();`

- Notice the word `String` written inside angled brackets <>

- This specifies that the `ArrayList` can hold `String` objects.

- If we try to store any other type of object in this `ArrayList`, an error will occur.

Using an `ArrayList`

- To populate the `ArrayList`, use the `add` method:
  - `nameList.add("James");`
  - `nameList.add("Catherine");`

- To get the current size, call the `size` method
  - `nameList.size();   // returns 2`

- To access items in an `ArrayList`, use the `get` method
- `nameList.get(1);`

Using an `ArrayList`

- The `ArrayList` class's `toString` method returns a string representing all items in the `ArrayList`
- `System.out.println(nameList);`
- This statement yields :
- `[ James, Catherine ]`

- The `ArrayList` class's `remove` method removes designated item from the
- `ArrayList`
- `nameList.remove(1);`
- This statement removes the second item.

Using an `ArrayList`

- The `ArrayList` class's `add` method with one argument adds new items to the end of the `ArrayList`
- To insert items at a location of choice, use the `add` method with two arguments:

- `nameList.add(1, "Mary");`
- This statement inserts the `String` "Mary" at index 1

- To replace an existing item, use the `set` method:
- `nameList.set(1, "Becky");`
- This statement replaces “Mary” with “Becky”

Using an `ArrayList`

- An `ArrayList` has a capacity, which is the number of items it can hold without increasing its size.
- The default capacity of an `ArrayList` is 10 items.
- To designate a different capacity, use a parameterized constructor:
- `ArrayList<String> list = new ArrayList<String>(100);`

Using an **`ArrayList`**

- You can store any type of *object* in an `ArrayList`

```
ArrayList<BankAccount> accountList =
                    new ArrayList<BankAccount>();
```

This creates an `ArrayList` that can hold `BankAccount` objects.

Using an **`ArrayList`**

- `// Create an ArrayList to hold BankAccount objects. ArrayList<BankAccount> list = new ArrayList<BankAccount>();`
- `// Add three BankAccount objects to the ArrayList. list.add(new BankAccount(100.0));`
- `list.add(new BankAccount(500.0)); list.add(new BankAccount(1500.0));`
- `// Display each item.`
- `for (int index = 0; index < list.size(); index++)`
- `{`
- `BankAccount account = list.get(index); System.out.println("Account at index " + index +`
    - `"\nBalance: " + account.getBalance());`
- `}`