2 Exceptions and Assertions

Topics

- What are Exceptions?
  - Introduction
  - The *Error* and *Exception* Classes
  - An Example
- Catching Exceptions
  - The *try-catch* Statements
  - The *finally* Keyword

Topics

- Throwing Exceptions
  - The *throw* Keyword
  - The *throws* Keyword
- Exception Categories
  - Exception Classes and Heirarchy
  - Checked and Unchecked Exceptions
  - User-Defined Exceptions

Topics

- Assertions
  - What are Assertions?
  - Enabling and Disabling Assertions
  - Assert Syntax

What are Exceptions?

- Definition:
  - Exceptional events
  - Errors that occur during runtime
  - Cause normal program flow to be disrupted
  - Examples
    - Divide by zero errors
    - Accessing the elements of an array beyond its range
    - Invalid input
    - Hard disk crash
    - Opening a non-existent file
    - Heap memory exhausted

The Error and Exception Classes

- *Throwable* class
  - Root class of exception classes
  - Immediate subclasses
    - *Error*
    - *Exception*

- Exception class
  - Conditions that user programs can reasonably deal with
  - Usually the result of some flaws in the user program code
  - Examples
    - Division by zero error
    - Array out-of-bounds error

The Error and Exception Classes

- Error class
  - Used by the Java run-time system to handle errors occurring in the run-time environment
  - Generally beyond the control of user programs
  - Examples
    - Out of memory errors
    - Hard disk crash

Exception Example

```
class DivByZero {
    public static void main(String args[]) {
        System.out.println(3/0);
        System.out.println(“Pls. print me.”);
    }
}
```

Exception Example

- Displays this error message

```
Exception in thread "main"
  java.lang.ArithmeticException: / by zero

      at DivByZero.main(DivByZero.java:3)
```

- Default handler
  - Prints out exception description
  - Prints the stack trace
    - Hierarchy of methods where the exception occurred
  - Causes the program to terminate

Catching Exceptions: The *try-catch* Statements

- Syntax:

```
try {
   <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
   <handler if ExceptionType1 occurs>
}
...
} catch (<ExceptionTypeN> <ObjName>) {
   <handler if ExceptionTypeN occurs>
}
```

Catching Exceptions: The *try-catch* Statements

```
class DivByZero {
   public static void main(String args[]) {
      try {
         System.out.println(3/0);
         System.out.println("Please print me.");
      } catch (ArithmeticException exc) {
         //Division by zero is an ArithmeticException
         System.out.println(exc);
      }
      System.out.println("After exception.");
   }
}
```

Catching Exceptions: The *try-catch* Statements

- Multiple catch example

```
class MultipleCatch {
   public static void main(String args[]) {
      try {
         int den = Integer.parseInt(args[0]);
         System.out.println(3/den);
      } catch (ArithmeticException exc) {
         System.out.println("Divisor was 0.");
      } catch (ArrayIndexOutOfBoundsException exc2) {
         System.out.println("Missing argument.");
      }
      System.out.println("After exception.");
   }
}
```

Catching Exceptions: The *try-catch* Statements

- Nested *try*s

```
class NestedTryDemo {
   public static void main(String args[]){
      try {
         int a = Integer.parseInt(args[0]);
         try {
            int b = Integer.parseInt(args[1]);
            System.out.println(a/b);
         } catch (ArithmeticException e) {
            System.out.println("Div by zero error!");
         }
         //continued...
```

Catching Exceptions: The *try-catch* Statements

```
        } catch (ArrayIndexOutOfBoundsException) {
            System.out.println("Need 2 parameters!");
        }
    }
}
```

Catching Exceptions: The *try-catch* Statements

- Nested *try*s with methods

```
class NestedTryDemo2 {
   static void nestedTry(String args[]) {
      try {
         int a = Integer.parseInt(args[0]);
         int b = Integer.parseInt(args[1]);
         System.out.println(a/b);
      } catch (ArithmeticException e) {
         System.out.println("Div by zero error!");
      }
   }
//continued...
```

Catching Exceptions: The *try-catch* Statements

```
    public static void main(String args[]){
        try {
            nestedTry(args);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Need 2 parameters!");
        }
    }
}
```

Catching Exceptions: The *finally* Keyword

- Syntax:

```
try {
   <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
   <handler if ExceptionType1 occurs>
} ...
} finally {
   <code to be executed before the try block ends>
}
```

- Contains the code for cleaning up after a try or a catch

Catching Exceptions: The *finally* Keyword

- Block of code is always executed despite of different scenarios:
  - Forced exit occurs using a *return*, a *continue* or a *break* statement
  - Normal completion
  - Caught exception thrown
    - Exception was thrown and caught in the method
  - Uncaught exception thrown
    - Exception thrown was not specified in any catch block in the method

Catching Exceptions: The *finally* Keyword

```
class FinallyDemo {
    static void myMethod(int n) throws Exception{
        try {
            switch(n) {
                case 1: System.out.println("1st case");
                        return;
                case 3: System.out.println("3rd case");
                        throw new RuntimeException("3!");
                case 4: System.out.println("4th case");
                        throw new Exception("4!");
                case 2: System.out.println("2nd case");
            }
//continued...
```

Catching Exceptions: The *finally* Keyword

```java
		} catch (RuntimeException e) {
				System.out.print("RuntimeException: ");
				System.out.println(e.getMessage());
		} finally {
				System.out.println("try-block entered.");
		}
	}
//continued...
```

Catching Exceptions: The *finally* Keyword

```
    public static void main(String args[]){
        for (int i=1; i<=4; i++) {
            try {
                FinallyDemo.myMethod(i);
            } catch (Exception e){
                System.out.print("Exception caught: ");
                System.out.println(e.getMessage());
            }
            System.out.println();
        }
    }
}
```

Throwing Exceptions: The *throw* Keyword

- Java allows you to throw exceptions:

```
throw <exception object>;
```

- Example:

```
throw new ArithmeticException("testing...");
```

Throwing Exceptions: The *throw* Keyword

```
class ThrowDemo {
    public static void main(String args[]){
        String input = “invalid input”;
        try {
            if (input.equals(“invalid input”)) {
                throw new RuntimeException("throw demo");
            } else {
                System.out.println(input);
            }
            System.out.println("After throwing");
        } catch (RuntimeException e) {
            System.out.println("Exception caught:" + e);
        }
    }
}
```

Throwing Exceptions: The *throws* Keyword

- A method is required to either catch or list all exceptions it might throw
  - Except for *Error* or *RuntimeException*, or their subclasses

- If a method may cause an exception to occur but does not catch it, then it must say so using the *throws* keyword
  - Applies to checked exceptions only

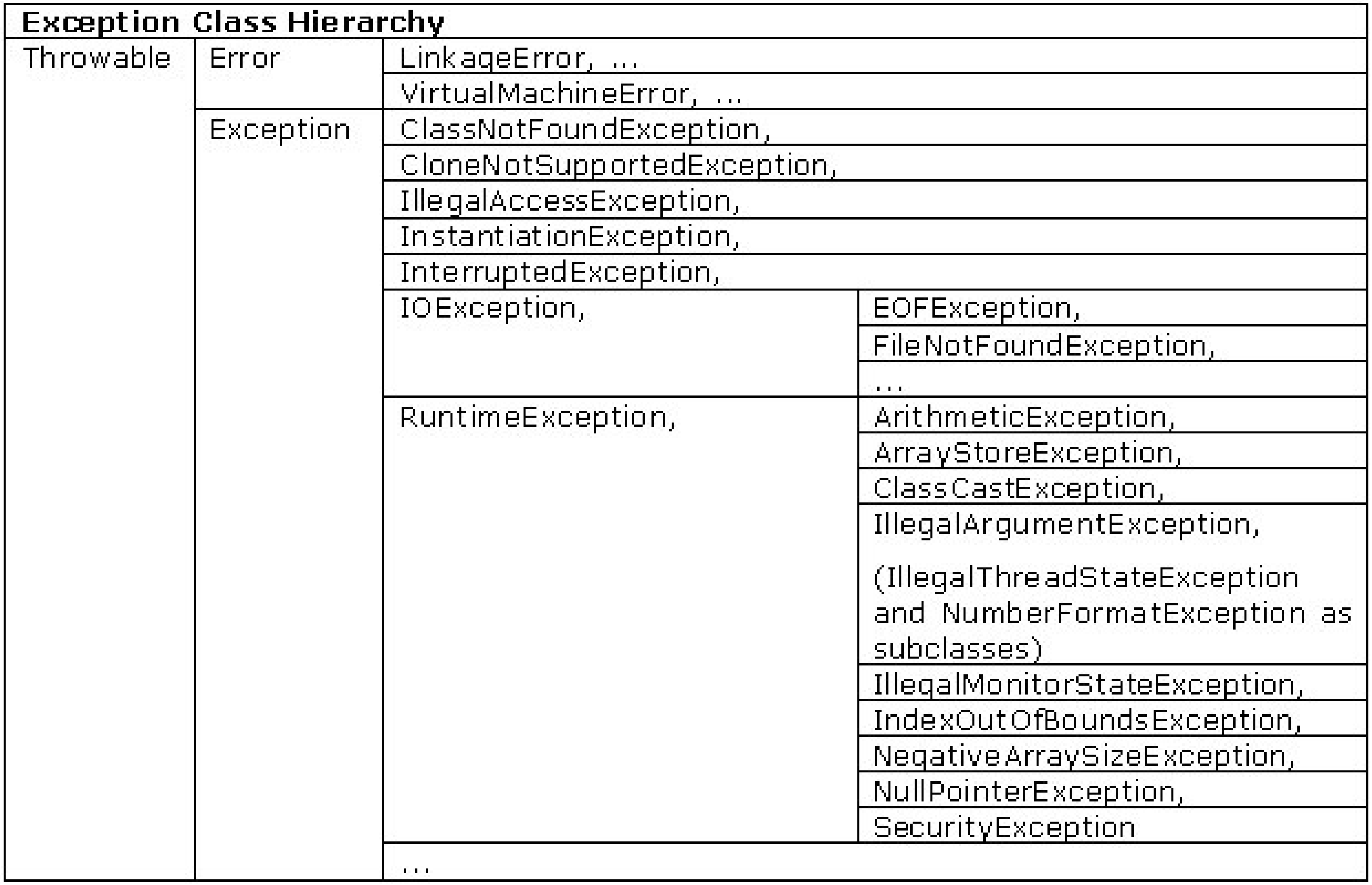
- Syntax:

```
<type> <methodName> (<parameterList>) throws
  <exceptionList> {
    <methodBody>
}
```

Throwing Exceptions: The *throws* Keyword

```
class ThrowingClass {
   static void meth() throws ClassNotFoundException {
      throw new ClassNotFoundException ("demo");
   }
}
class ThrowsDemo {
   public static void main(String args[]) {
      try {
         ThrowingClass.meth();
      } catch (ClassNotFoundException e) {
         System.out.println(e);
      }
   }
}
```

Exception Classes and Hierarchy



Exception Class Hierarchy			
Throwable	Error	LinkageError, ...	
		VirtualMachineError, ...	
	Exception	ClassNotFoundException,	
		CloneNotSupportedException,	
		IllegalAccessException,	
		InstantiationException,	
		InterruptedException,	
		IOException,	EOFException,
			FileNotFoundException,
			...
		RuntimeException,	ArithmeticException,
			ArrayStoreException,
			ClassCastException,
			IllegalArgumentException, (IllegalThreadStateException and NumberFormatException as subclasses)
			IllegalMonitorStateException,
			IndexOutOfBoundsException,
			NegativeArraySizeException,
			NullPointerException,
			SecurityException
		...	

Exception Classes and Hierarchy

- Multiple catches should be ordered from subclass to superclass.

```
class MultipleCatchError {
    public static void main(String args[]){
        try {
            int a = Integer.parseInt(args [0]);
            int b = Integer.parseInt(args [1]);
            System.out.println(a/b);
        } catch (Exception ex) {
        } catch (ArrayIndexOutOfBoundsException e) {
        }
    }
}
```

Checked and Unchecked Exceptions

- Checked exception
  - Java compiler checks if the program either catches or lists the occurring checked exception
  - If not, compiler error will occur
- Unchecked exceptions
  - Not subject to compile-time checking for exception handling
  - Built-in unchecked exception classes
    - Error
    - RuntimeException
    - Their subclasses
  - Handling all these exceptions may make the program cluttered and may become a nuisance

User-Defined Exceptions

- Creating your own exceptions
  - Create a class that *extends* the *RuntimeException* or the *Exception* class
  - Customize the class
    - Members and constructors may be added to the class

- Example:

```
class HateStringExp extends RuntimeException {
    /* No longer add any member or constructor */
}
```

User-Defined Exceptions

- Using user-defined exceptions

```
class TestHateString {
    public static void main(String args[]) {
        String input = "invalid input";
            try {
                if (input.equals("invalid input")) {
                    throw new HateStringExp();
                }
                System.out.println("Accept string.");
            } catch (HateStringExp e) {
                System.out.println("Hate string!”);
            }
    }
}
```

What are Assertions?

- Allow the programmer to find out if an assumption was met
  - Example: month
- Extension of comments wherein the assert statement informs the person reading the code that a particular condition should always be satisfied
  - Running the program informs you if assertions made are true or not
  - If an assertion is not true, an *AssertionError* is thrown
- User has the option to turn it off or on at runtime

Enabling or Disabling Assertions

- Program with assertions may not work properly if used by clients not aware that assertions were used in the code

- Compiling
  - With assertion feature:

```
javac -source 1.4 MyProgram.java
```

  - Without the assertion feature:

```
javac MyProgram.java
```

- Enabling assertions:
  - Use the –enableassertions or –ea switch.

```
java -enableassertions MyProgram
```

Assert Syntax

- Two forms:
  - Simpler form:

    ```
    assert <expression1>;
    ```

    where

    - <expression1> is the condition asserted to be true
  - Other form:

    ```
    assert <expression1> : <expression2>;
    ```

    where

    - <expression1> is the condition asserted to be true
    - <expression2> is some information helpful in diagnosing why the statement failed

Assert Syntax

```
class AgeAssert {
   public static void main(String args[]) {
      int age = Integer.parseInt(args[0]);
      assert(age>0);
      /* if age is valid (i.e., age>0) */
      if (age >= 18) {
         System.out.println("You're an adult! =)");
      }
   }
}
```

Summary

- Exceptions
  - Definition
  - *try*, *catch* and *finally*
  - *throw* and *throws*
  - *Throwable*, *Exception*, *Error* classes
  - Checked and Unchecked Exceptions
  - User-Defined Exceptions
- Assertions
  - Definition
  - Enabling and Disabling Assertions
  - Assert Syntax