Programming in Java

Exception Handling

Exception Handling

- An *exception* is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error.

- In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome.

- Java's exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world.

Exception-Handling

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by our code.

Exception-Handling

- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Program statements that we want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown.
- Our code can catch this exception (using **catch**) and handle it in some rational manner.
- System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

general form of an exception-handling block

```
try {
    // block of code to monitor for errors
    }
    catch (ExceptionType1 exOb) {
                // exception handler for ExceptionType1
                }
    catch (ExceptionType2 exOb) {
                // exception handler for ExceptionType2
                }
                // ...
    finally {
                // block of code to be executed after try block ends
               }
```

Exception Types

- All exception types are subclasses of the built-in class **Throwable**.
- Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that we will subclass to create our own custom exception types.
- There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that we write and include things such as division by zero and invalid array indexing.
- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by our program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.
- Stack overflow is an example of such an error.

Uncaught Exceptions

- ```
  class Exc0 {
      public static void main(String args[]) {
          int d = 0;
          int a = 42 / d;
                                  }
  }
  ```
- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.
- Any exception that is not caught by our program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

  Here is the exception generated when this example is executed:

  ```
  java.lang.ArithmeticException: / by zero
  at Exc0.main(Exc0.java:4)
  ```
  ```

Uncaught Exceptions

- ```
  class Exc1 {
   static void subroutine() {
            int d = 0;
            int a = 10 / d;
                  }
   public static void main(String args[]) {
            Exc1.subroutine();
                  }
        }
  ```
- The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

  ```
  java.lang.ArithmeticException: / by zero
  at Exc1.subroutine(Exc1.java:4)
  at Exc1.main(Exc1.java:7)
  ```
  ```

Using try and catch

- Handling exception by our self it allows us to fix the error and it prevents the program from automatically terminating.

- To guard against and handle a run-time error, simply enclose the code that we want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that we wish to catch.

Using try and catch

```
class Exc2 {
        public static void main(String args[]) {
                int d, a;
                try { // monitor a block of code.
                    d = 0;
                    a = 42 / d;
                   System.out.println("This will not be printed.");
      }
   catch (ArithmeticException e) {       // catch divide-by-zero error
            System.out.println("Division by zero.");
       }
   System.out.println("After catch statement.");
              }
    }
```

Using try and catch

- A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.

- A **catch** statement cannot catch an exception thrown by another **try** statement. The statements that are protected by **try** must be surrounded by curly braces.

- The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

Displaying a Description of an Exception

- We can display the description of exception in a println( ) statement by simply passing the exception as an argument.
- For example, the **catch** block rewritten like this:

```
catch (ArithmeticException e) {
        System.out.println("Exception: " + e);
        a = 0;          // set a to zero and continue
        }
```

- When this version is substituted in the program, and the program is run, each divide-by zero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

Multiple catch Clauses

- If more than one exception can be raised by a single piece of code then we can specify two or more **catch** clauses, each catching a different type of exception.

- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch** block.

Demonstrate multiple catch statements.

```
class MultiCatch {
        public static void main(String args[]) {
                try {
                        int a = args.length;
                        System.out.println("a = " + a);
                        int b = 42 / a;
                        int c[] = { 1 };
                        c[42] = 99;
                } catch(ArithmeticException e)
                        {
                        System.out.println("Divide by 0: " + e);
                        }
                catch(ArrayIndexOutOfBoundsException e)
                        {
                                System.out.println("Array index oob: " + e);
                        }
```

Demonstrate multiple catch statements.

C:\>java MultiCatch

a = 0

Divide by 0: java.lang.ArithmeticException: / by zero

After try/catch blocks.

C:\>java MultiCatch TestArg

a = 1

Array index oob: java.lang.ArrayIndexOutOfBoundsException:42

After try/catch blocks.

Demonstrate multiple catch statements.

- When we use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their super classes.

- This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

Demonstrate multiple catch statements.

```
class SuperSubCatch {
        public static void main(String args[]) {
                try {
                        int a = 0;
                        int b = 42 / a;
                }
                 catch(Exception e) {
                System.out.println("Generic Exception catch.");
                                        }
    /* This catch is never reached becauseArithmeticException is a subclass of
Exception. */
        catch(ArithmeticException e) { // ERROR - unreachable
                   System.out.println("This is never reached.");
                }
                }
        }
```

Nested try Statements

- The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**.
- Each time a **try** statement is entered, the context of that exception is pushed on the stack.
- If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match.
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

Nested try Statements

```
class NestTry {
public static void main(String args[]) {
try {
        int a = args.length;
/* If no command-line args are present, the following statement will generate
a divide-by-zero exception. */
        int b = 42 / a;
        System.out.println("a = " + a);
        try { // nested try block
/* If one command-line arg is used, then a divide-by-zero exception will be
generated by the following code. */
            if(a==1) a = a/(a-a); // division by zero
```

Nested try Statements

```
/* If two command-line args are used, then generate an out-of-bounds exception. */
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

Nested try Statements

C:\>java NestTry

Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One

a = 1

Divide by 0: java.lang.ArithmeticException: / by zero

C:\>java NestTry One Two

a = 2

Array index out-of-bounds:

java.lang.ArrayIndexOutOfBoundsException:42

throw

- It is possible for our program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

  throw *ThrowableInstance;*

- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- There are two ways we can obtain a **Throwable** object:
- using a parameter in a **catch** clause, or creating one with the **new** operator.
- The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed.
- The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement.
- If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

throw

```
// Demonstrate throw.
class ThrowDemo {
        static void demoproc() {
                try {
                        throw new NullPointerException("demo");
                    } catch(NullPointerException e) {
                        System.out.println("Caught inside demoproc.");
                            throw e; // rethrow the exception
                                }
                }
public static void main(String args[]) {
        try {
                demoproc();
                } catch(NullPointerException e) {
                        System.out.println("Recaught: " + e);
                                                }
                                        }
        }
```

throw

throw new NullPointerException("demo");

- **new** is used to construct an instance of **NullPointerException**.
- Many of Java's built in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter.
- When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**.

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception.
- We do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw.
- This is the general form of a method declaration that includes a **throws** clause:

type method-name(*parameter-list*) throws *exception-list*

{

// body of method

}

- Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

throws

```
// This program contains an error and will not compile.
class ThrowsDemo
    {
        static void throwOne()
                {
                    System.out.println("Inside throwOne.");
                     throw new IllegalAccessException("demo");
                }
        public static void main(String args[])
            {
                throwOne();
            }
     }
```

- This example is incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

throws

```
// This is now correct.
class ThrowsDemo {
        static void throwOne() throws IllegalAccessException
          {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
          }
public static void main(String args[]) {
        try {
                throwOne();
           } catch (IllegalAccessException e) {
                System.out.println("Caught " + e);
                }
           }
  }
```

- **main( )** must define a **try/catch** statement that catches this exception.

finally

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.

- It is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.

- For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this issue.

finally

- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.

- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

finally

```
// Demonstrate finally.
class FinallyDemo {          // Through an exception out of the method.
        static void procA() {
                try {
                        System.out.println("inside procA");
                        throw new RuntimeException("demo");
                } finally {
                                System.out.println("procA's finally");
                 }
                }
// Return from within a try block.
        static void procB() {
                try {
                        System.out.println("inside procB");
                        return;
                } finally {
                                System.out.println("procB's finally");
                           }
                        }
```

finally

```
// Execute a try block normally.
static void procC() {
        try {
                System.out.println("inside procC");
            } finally {
                    System.out.println("procC's finally");
                }
            }
public static void main(String args[]) {
        try {
                procA();
            } catch (Exception e) {
                    System.out.println("Exception caught");
                }
            procB();
            procC();
        }       }
```

Java's Built-in Exceptions

- Inside the standard package **java.lang**, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type **RuntimeException**.

- These exceptions need not be included in any method's **throws** list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions.

- The exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*.

- Java defines several other types of exceptions that relate to its various class libraries.

Java's Unchecked RuntimeException Subclasses Defined in java.lang

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Java's Checked Exceptions Defined in java.lang

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Creating Own Exception Subclasses

```
class MyException extends Exception {
        private int detail;
        MyException(int a) {
                detail = a;
                }
        public String toString() {
                return "MyException[" + detail + "]";
                }       }
class ExceptionDemo {
        static void compute(int a) throws MyException {
                System.out.println("Called compute(" + a + ")");
                    if(a > 10)
                            throw new MyException(a);
                System.out.println("Normal exit");
                }
        public static void main(String args[]) {
                try {
                            compute(1);
                            compute(20);
                     } catch (MyException e) {
                System.out.println("Caught " + e);
        }      }          }
```

```
import java.util.*;
class NegativeCircleRadiusException extends
Exception
{
public String toString()
{
return "Radius of Circle cannot be nagative";
}
}
class Circle
{
double radius;

Circle(double r)
{
radius=r;
}
double area() throws
NegativeCircleRadiusException
{
if(radius<0.0)
{
throw new NegativeCircleRadiusException();
}
else
return 3.14*radius*radius;

}
}
```

```
class TestNegativeRadius
{
public static void main(String[] args)
{
Scanner sc= new Scanner(System.in);
System.out.println("Enter Radius of Circle");
double x=sc.nextDouble();
Circle c1=new Circle(x);
try
{
double d=c1.area();
System.out.println("Area of Circle is"+d);
}
catch(NegativeCircleRadiusException e)
{
e. printStackTrace();
}
}
}
```

- (**NumberFormatException**) Write a program that prompts the user to read two integers and displays their sum. Your program should prompt the user to read the number again if the input is incorrect .

- Write a program that prompts the user to read radius of circle and display the area of circle if radius given by user is positive otherwise throw NegativeCircleRadiusException.

- (**IllegalTriangleException**) Exercise defined the **Triangle** class with three sides. In a triangle, the sum of any two sides is greater than the other side. The **Triangle** class must adhere to this rule. Create the **IllegalTriangleException** class, and modify the constructor of the **Triangle** class to throw an **IllegalTriangleException** object if a triangle is created with sides that violate.

- the rule, as follows:

```
/** Construct a triangle with the specified sides */
public Triangle(double side1, double side2, double side3)
throws IllegalTriangleException {
// Implement it
}
```

Assertions

- An *assertion* is a statement in the Java programming language that enables you to test your assumptions about your program.
- For example, you might have a method that should always return a positive integer value.

- You might test this by asserting that the return value is greater than zero using an assert statement. At run time, if the condition is true, no other action takes place. However, if the condition is false, then an AssertionError is thrown.

- An assertion uses **assert** keyword.

- The **assert** keyword has two forms. The first is shown here:
  assert *condition*;

where

- *condition* is a boolean expression. When the system runs the assertion, it evaluates *condition* and if it is false throws an AssertionError with no detail message.

- The second form of **assert** is shown here:
  assert *condition: expr;*

where:

- *Condition* is a boolean expression.
- *expr* is an expression that has a value. (It cannot be an invocation of a method that is declared void.)
- Use this version of the assert statement to provide a detail message for the AssertionError. The system passes the value of *expr* to the appropriate AssertionError constructor, which uses the string representation of the value as the error's detail message.