Exception Handling in Java (Contd.)

Rethrowing Exceptions

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

The statement *throw ex* rethrows the exception so that other handlers get a chance to process the exception *ex*.

Sometimes you may need to throw a new exception with additional information along with the original exception. This is called *chained exceptions*.

The `finally` Clause

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

Code in the *finally* block is executed under all circumstances, regardless of whether an exception occurs in the *try* block or is caught.

Trace a Program Execution

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

Next statement;
```

Suppose no exceptions in the statements

Trace a Program Execution

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

The final block is always executed

```
Next statement;
```

Trace a Program Execution

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

```
Next statement;
```

Next statement in the method is executed

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}
```

The exception is handled.

```
Next statement;
```

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}
```

```
Next statement;
```

The next statement in the method is now executed.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}
Next statement;
```

statement2 throws an exception of type Exception2.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}
Next statement;
```

Handling exception

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}
Next statement;
```

Execute the final block

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}
Next statement;
```

Rethrow the exception and control is transferred to the caller

Cautions When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
- Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.

When to Throw Exceptions

- An exception occurs in a method.
- If you want the exception to be processed by its caller, you should create an exception object and throw it.
- If you can handle the exception in the method where it occurs, there is no need to throw it.

When to Use Exceptions

When should you use the try-catch block in the code?

You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code:

```
try {

  System.out.println(refVar.toString());

}

catch (NullPointerException ex) {

  System.out.println("refVar is null");

}
```

When to Use Exceptions

is better to be replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

Creating Custom Exception Classes

- Use the exception classes in the API whenever possible.
- Create custom exception classes if the predefined classes are not sufficient.
- Declare custom exception classes by extending Exception or a subclass of Exception.

Example 5 - Custom Exception Class (*InvalidRadiusException.java, CircleWithRadiusException.java*)

In previous example, the setRadius method throws an exception if the radius is negative.

Suppose you wish to pass the radius to the handler, you have to create a custom exception class.