Exception Handling
Part 3

Prof. Siddharth Shah
Department of Computer Engineering
Dharmsinh Desai University

Outline

- Checked and Unchecked Exceptions
- Custom Exception
- Chained Exceptions

Checked and Unchecked Exceptions

- **Checked** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword. e.g. **IOException**
- **Unchecked** are the exceptions that are not checked at compiled time, e.g. **NullPointerException**
- Exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under Throwable is checked.

Creating Custom Exception - 1

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable).
- Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

Creating Custom Exception - 2

- The Exception class does not define any methods of its own.
- It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.
- Exception defines four public constructors. Two support chained exceptions, described in the next section. The other two are shown here:
- **Exception( ):** creates an exception that has no description.
- **Exception(String msg):** lets you specify a description of the exception.

Chained Exceptions - 1

- The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception.
- For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an **I/O error** occurred, which caused the divisor to be set improperly.
- Although the method must certainly throw an ArithmeticException, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error.
- Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

Chained Exceptions - 2

- To allow chained exceptions, two constructors and two methods were added to Throwable.
- The constructors are shown here:
- **Throwable(Throwable causeExc):** causeExc is the exception that causes the current exception. That is, causeExc is the underlying reason that an exception occurred.
- **Throwable(String msg, Throwable causeExc):** this form allows you to specify a description at the same time that you specify a cause exception.
- These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

Chained Exceptions - 3

- The chained exception methods supported by Throwable are getCause( ) and initCause( ).
- **Throwable getCause( ):** returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
- **Throwable initCause(Throwable causeExc):** associates causeExc with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created.
- However, the cause exception can be set only once. Thus, you can call initCause( ) only once for each exception object.
- Furthermore, if the cause exception was set by a constructor, then you can't set it again using initCause( ).