

CS202 Java Object Oriented Programming

Errors and Exception Handling

Chengyu Sun
California State University, Los Angeles

Examples of Exceptions

◆ArrayIndexOutOfBoundsException

```
int a[] = new int[10];  
for( int i=0 ; i <= a.length ; ++i ) a[i] = i;
```

◆NumberFormatException

```
String s = "7.3";  
int n = Integer.parseInt( s );
```

Runtime Errors

- ◆Errors happened when the program is running, e.g.
 - Access an element outside array boundary
 - Read a file which doesn't exist
 - User input errors
- ◆Runtime error handling
 - Error codes
 - Exceptions

Error Codes

```
// n >= 1  
int factorial( int n )  
{  
    // return an error code  
    if( n < 1 ) return -1;  
  
    int fact = 1;  
    for( int i=1 ; i <= n ; ++i )  
        fact *= i;  
  
    return fact;  
}  
  
int f = factorial(x);  
if( f == -1 ) // error handling  
{  
    System.err.println( "error!" );  
    System.exit(1);  
}  
else // regular code  
{  
    // do something with f  
}
```

Raise An Exception – throw

```
int factorial( int n ) throws LessThanOneException  
{  
    if( n < 1 ) throw new LessThanOneException();  
  
    int fact = 1;  
    for( int i=1 ; i <= n ; ++i ) fact *= i;  
    return fact;  
}
```

A Closer Look at Throw

The diagram illustrates the execution of the `factorial` method. It shows two paths from the `if( n < 1 )` condition:

- Exception path:** If `n < 1`, an `Exception value` (a new `LessThanOneException()`) is created and thrown. This path is labeled **Exception type** and **Exception value**.
- Return path:** If `n >= 1`, the method calculates the factorial and returns it. This path is labeled **Returned value**.

The `int factorial( int n )` signature is labeled **Return type**.

Exception Class

- ◆ Pre-defined in Java
- ◆ All exception classes must inherit from this class
- ◆ Important methods
 - `Exception()`
 - `Exception( String msg )`
 - `String getMessage()`
 - `void printStackTrace()`

Create An Exception Class

```
public class LessThanOneException extends Exception {  
    public LessThanOneException() { super(); }  
    public LessThanOneException( String msg )  
    {  
        super(msg);  
    }  
}
```

Handle An Exception – try and catch

```
try    // regular code  
{  
    int f = factorial(n);  
    // do something with f  
}  
catch( LessThanOneException e ) // error handling  
{  
    System.err.println( "error!" );  
    System.exit(1);  
}
```

Using Exceptions

- ◆ In a method where an error may occur
 - Declare exception type in method header: **throws**
 - Raise an exception if the error occurs: **throw**
- ◆ In the calling method
 - Handle the exception
 - Enclose regular code in a **try** block
 - Enclose error handling code in a **catch** block
 - Do *not* handle the exception
 - Declare the exception in method header

Throw Multiple Exceptions

```
public void foo() throws ExceptionA, ExceptionB, ExceptionC  
{  
    ...  
    throw new ExceptionA("Bad thing A");  
    ...  
    throw new ExceptionB("Bad Thing B");  
    ...  
    throw new ExceptionC("Bad Thing C");  
}
```

Catch Multiple Exceptions

```
try  
{  
    foo();  
}  
catch( ExceptionA ea )  
{  
    // do something  
}  
catch( ExceptionB eb )  
{  
    // do something  
}  
catch( ExceptionC ec )  
{  
    // do something  
}
```

◆ Or, if we don't need to distinguish exactly which type of exception occurs

```
try  
{  
    foo();  
}  
catch( Exception e )  
{  
    System.err.println( e.getMessage() );  
    System.exit(1);  
}
```

A More Complex Example

```
readFile {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
}
```

Handling Errors with Error Code

```
errorCodeType readFile {  
    errorCode = 0;  
    open the file;  
    if (theFileIsOpen) {  
        determine the length of the file;  
        if (gotTheFileLength) {  
            allocate that much memory;  
            if (gotEnoughMemory) {  
                read the file into memory;  
                if (readFailed) { errorCode = -1; }  
            }  
            else { errorCode = -2; }  
        } else { errorCode = -3; }  
        close the file;  
        if (theFileDintClose && errorCode == 0) { errorCode = -4; }  
        else { errorCode = errorCode and -4; }  
    }  
    else { errorCode = -5; }  
    return errorCode;  
}
```

Handling Errors with Exceptions

```
readFile {  
    try  
    {  
        open the file;  
        determine its size;  
        allocate that much memory;  
        read the file into memory;  
        close the file;  
    }  
    catch (fileOpenFailed) { doSomething; }  
    catch (sizeDeterminationFailed) { doSomething; }  
    catch (memoryAllocationFailed) { doSomething; }  
    catch (readFailed) { doSomething; }  
    catch (fileCloseFailed) { doSomething; }  
}
```

Advantages of Exceptions

- ◆ Separate error handling code with from regular code
- ◆ Propagate errors up the call stack
- ◆ Group error types and error differentiation

Error Propagation Example

```
method1 {  
    try { call method2; }  
    catch (exception) { doErrorProcessing; }  
}  
  
method2 throws exception { call method3; }  
  
method3 throws exception { call readFile; }
```

Error Grouping and Differentiation Example

