

Exceptions

Slides derived from the work of Dr. Amy McGovern and Dr. Deborah Trytten

Challenges to Building Robust Software

Software will often be used in conditions that you (the designer) cannot precisely anticipate

- A user might enter data that is incorrect
- A user programmer might create or manipulate objects in incorrect ways
- Programmatic operations might fail
- System errors might occur

We need to address all of these

Techniques for Highlighting Errors

Techniques for Highlighting Errors

When an error in a method occurs, we can use:

- `System.out.println()`
 - Create a log file that keeps track of every important event/operation inside your code (including errors). Log4J is a tool for doing this...
- Return a value that indicates an error
- `System.exit()`: halt execution of your program

Techniques for Highlighting Errors

- `System.out.println()`
 - Create a log file that keeps track of every important event/operation inside your code (including errors). Log4J is a tool for doing this...
- Return a value that indicates an error
- `System.exit()`: halt execution of your program

None of these techniques allows a program to robustly take corrective action

Exceptions

- You have already seen some exceptions...

Exceptions

- You have already seen some exceptions...
 - NullPointerException
 - IndexOutOfBoundsException
 - IOException
- These cause your program to halt
 - Show the file and line number where the failure occurred
 - Show the “stack trace” – the nested list of method calls that has brought us to the failure

Examples

- Operations on uninstantiated objects
- Creating arrays of negative size

Example Analysis

In these examples:

- The exception causes a halt to the program
- In our array size example:
 - Entry of data happens in one method
 - But the program halts in another method. These can be very far apart in large programs
- Want our code to address as many errors as possible

Syntax

```
try
{
    // some code that could throw an exception
}
catch (ExceptionName e)
{
    // Fix the problem here
}
```

If you're not going to fix the problem there is no reason to catch the Exception

Throwing Exceptions

When an Exception is thrown:

- Execution stops immediately
- JVM examines the call stack for a matching ***catch*** statement
- Execution continues within the body of the matching catch statement
- Then: execution continues after the try-catch

Throwing Exceptions

- The catch statement should be placed at a point where the program can address the issue (we call this *handling the exception*)
- Exceptions not caught within the method cause the search to continue in the next higher method
- Exceptions not caught by any method in the call stack cause the program to halt

Hierarchy of Exception Classes

Object

- Throwable
 - Exception
 - RuntimeException
 - NullPointerException
 - ArithmeticException
 - ArrayIndexOutOfBoundsException
 - IOException
 - Error

Error Class

- See API
- When these occur, we generally accept that we cannot recover
- Instead, we try to gracefully clean things up before halting
 - Save critical data to files
 - Alert the user

RuntimeException

Most often caused by programming errors

- **Unchecked exception**: we don't have to explicitly address these in code unless we want to (so, no “throws” statements necessary)
- Usually a sign that we need to debug our code
- Example: ArrayList API (get, set)

Not RuntimeExceptions

- All exceptions that are not RuntimeExceptions are **checked exceptions**
- Any method that can throw one of these exceptions must declare this in the method prototype (throws NameOfException)
- Any method that calls one of these methods:
 - Must address the exception with a try/catch
 - or also throw the exception

Creating Our Own Exceptions

- Only create a new exception if there isn't already one that does the job
- Extend an existing class
 - Often Exception or RuntimeException
 - Which is best?
- Implement the constructors
- Can add our own data!

Back to: Prompting the User for an Array Size

- Create an `IllegalSizeException`
- Prompt the user for a size
- If the given value is non-positive, then throw our new Exception
- When the Exception is received, then keep prompting for an array size

Back to: Prompting the User for an Array Size

What about when the user enters a non-number?

Back to: Prompting the User for an Array Size

What about when the user enters a non-number?

- We can also throw an exception under these conditions

Best Practices

- Don't overuse Exceptions:
 - If your code can detect and address the error right then and there, don't throw
 - Only use if interrupting the flow of the code is the right way to address the problem
- Only introduce new Exception classes when necessary
- If you are receiving unchecked exceptions, then address the bug – don't ask your code to recover

Multiple Catch Blocks

- Searched in order
- First one to match “wins” and no others are checked

```
try
{
}
catch (RunTimeException e)
{
}
catch (ArithmeticException e)
{
}
```

Finally Blocks: Cleaning Up

Guaranteed to execute finally block after the try block:

- Even if an exception occurs within the try
- Even if the try includes a “return”

```
try
{
}
catch (RunTimeException e)
{
}
finally
{
}
```

Rethrowing Exceptions

Sometimes your catch can't address the Exception

- Rethrow original Exception object
- or create a new one to throw...

```
try
{
}
catch (RunTimeException e)
{
    throw e;
    or
    throw new MyException("foo");
}
```

Examples: File I/O

- BufferedReader: see readLine()
- Double: see: valueOf(String)

