Java Programming: Introduction

Exceptions

Classical Error Handling

- Boolean functions
  - Return true or false
- Int functions
  - Return –1
- Other
  - Return args w/special values

Classical Error Handling Example

- In some programming schemes, error conditions are handled with returned error codes
- This is still done and is often a very good approach

```
int  success = openResource( String name )
if ( success == 0 ) {
     useResource( name );
}
else {
     cout << " open failed on " << name << endl;
     return;
}
```

What are the problems with this?

- Programmer must remember to check
- Awkward -> not a standard way to check
- Source bloat
  - Diminished readability
- Can be ignored

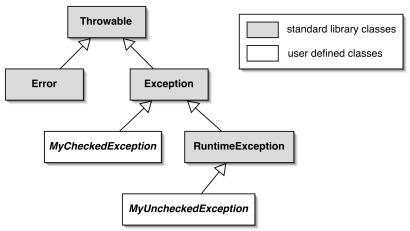
Java's Exceptions

- Java continues the tradition which C++ follows by making runtime error handling a part of the core API
- Enables transfer of control from where the error occurred to where it can be handled
- *try*, *catch*, *finally*, *throw* and *throws*
- Clean way to check for errors w/o cluttering source
- Can **NOT** be ignored
- No Silver Bullet

Defining Classes

- Thus, errors in Java are (not surprisingly) defined in classes
- All error-related classes inherit from java.lang.Throwable
- Here we see the two types of Throwable: Error and Exception

Java's Exception Hierarchy



java.lang.Error

- Reserved for fundamental, usually serious problems at the VM level
- ThreadDeath, LinkageError, VirtualMachineError
- Should not be caught--they can occur anywhere, so trying to handle them is impractical. Sometimes they are fatal anyway.
- Examples: VirtualMachineError:
  - OutOfMemoryError
- Examples: LinkageError:
  - ClassFormatError, NoClassDefFoundError
- It is very rare to define new Errors

java.lang.Exception

- Runtime errors that are usually recoverable
- It's common to define new ones
- Must be handled or code will not compile, except for instances of RuntimeException. RuntimeExceptions are like Errors, in that the problem is usually fundamental and potentially serious.
- It is possible to handle RuntimeExceptions, though again not mandatory. Whether advisable to do so depends on the context.

RuntimeException Examples

- NullPointerException - e.g. trying to invoke a method on an object reference that is null.
- IndexOutOfBoundsException - e.g. trying to access an array element or String element out of bounds.
- IllegalArgumentException - e.g. NumberFormatException where you try to convert a String to an incompatible numeric type.
- ClassCastException - e.g. Trying to cast to a class your object is not a representative of.
- ArithmeticException - e.g. Invalid arithmetic operation like integer divide by zero.

Brief Review

- So far, we've seen three types of error-related classes:
  - Errors
  - RuntimeExceptions
  - all other Exceptions

From now on, we'll focus on the third of these.

Defining New Exceptions

- Remember, Exceptions are classes just like all other classes. The same rules apply for defining them, and the same OO principles that apply to other classes you design apply to Exceptions as well.
- At a minimum, you should always provide a no-args constructor that sets a default message. The compiler doesn't enforce this, but it's a good idea.

Defining New Exceptions (cont.)

Example

```
public class AnException extends Exception {
  public AnException() {
   super("Bad news, something has happened.");
  }
}
```


Defining New Exceptions (cont.)

Of course, try to give Exceptions names that make sense. Think of situations in which the following Exceptions might be thrown:

FileNotFoundException
InvalidFormatException
MalformedURLException
MediumNotAvailableException
ColumnNotFoundException

Think for a moment about dealing with these situations in your favorite procedural language!

How do they occur ?

- All exceptions are generated by methods
- In fact, a method declares in its signature which exceptions (if any) it generates ("throws")
- Syntax:

```
public void aMethod() throws AnException {}
```

- If a method does not indicate that it throws an exception, it won't throw it. (Any method can throw a RuntimeException or Error, regardless of its signature.)

Handling Exceptions (***try***/***catch***)

- So, you can tell from looking at the javadoc API (or other class documentation that shows method signatures) whether you need to worry about handling exceptions.
- To handle exceptions, enclose the method invocation in a **`try/catch`** block.

Example (***try***/***catch***)

Example (note where curly braces appear):

```
public void myMethod() {
  try {
   aMethod();
   System.out.println("After aMethod()");
  } catch (AnException e) {
   System.err.println("Exception!");
  }
  System.out.println("Done.");
}
```


Handling Multiple Exceptions

What if there are multiple exceptions that could be thrown within a try{} block?

- Each exception will need a catch{} that can handle it.
- The VM will go through the list of catch{} blocks for the try{}, and will enter the first one that catches the same type of Exception that has been thrown.
- If you don't provide an appropriate catch{} for each exception that can be thrown, you will get a compile error.

Handling Multiple Exception (cont)

- You are not required to catch the exact Exception subclass that is thrown. You can also catch any superclass of the one that is thrown.
- So, if ExceptionB extends ExceptionA and ExceptionB is thrown, it is permissible to catch either ExceptionA or ExceptionB.
- It is usually advisable to be as specific as you can in what you catch. This encourages more appropriate error handling.
- Remember that the VM searches the list sequentially for the first acceptable catch{}!

Example of Multiple Exceptions

`MultipleExceptions.java`

Handling Exceptions (***finally***)

What if there was some code we wanted to execute regardless of whether aMethod() throws AnException?

- Use a **`finally`** block after the **`catch`** block:

```
… } catch(AnException e) {
    System.err.println("Exception!");
  } finally {
    System.out.println("Either way");
  } …
```

Summary of

try*/*catch*/*finally

Keyword ***try***

- *try* blocks are denoted by *try*{…}
- *try* blocks are executed until
  - Completes successfully
  - Exception is thrown

Keyword ***catch***

- Specify the type of exception object to catch
- What to do when you catch it?
  - Handle and recover
  - Cleanup and rethrow
  - *catch(Exception e) {}*
    - Avoid this…why?

Keyword ***finally***

- Enables execution of code with or w/o exception being thrown
- Finally blocks are denoted by finally{…}
- Good for
  - Maintaining internal state
  - Freeing non object resources
  - Makes writing exceptions easier

Handling Exceptions(***throws***)

- If it is not appropriate for your method to handle an exception, simply rethrow it by adding a **`throws`** to your method's signature
- Returning to the earlier example:

```
public void myMethod() throws AnException {
  aMethod();
}
```

- At some point, the exception must be caught, so there's no way around the rules!

Handling Exceptions(***throw***)

- In classes you design, there will be occasions when it is appropriate for you to throw Exceptions. You can throw Exceptions defined in the Core API, or you can create your own. Generally, you'll create your own.
- Example:

```
if (bad_news)
  throw new AnException();
else {
  // continue with business of your method
}
```

Summary: Keyword *throw*

- The way exceptions are *thrown*
  - I.e. transferred
- Takes an object as its parameter
- Remember, Exceptions are objects
  - Must be created -> new Exceptions();

Summary: Keyword throws

- Advertises what exceptions are thrown by a method
- Should be as specific as possible

Summary of Exception Handling

`Chemist.java`

Hints/Tricks
Stack Trace/Message

- In a catch{}, you are passed an instance of the Exception thrown. You can use this object in your handler code.
- Throwable defines several useful methods:
  - getMessage() returns an appropriate message as a String
  - printStackTrace() dumps the series of method calls on the call stack
- Individual Exception subclasses usually add additional methods appropriate for the situation

Hints/Tricks
Stack trace/Message(cont)

`PrintStackTraceMsg.java`

Hints/Tricks
Using Exceptions in a normal way

`ExpandableArray.java`

Hints/Tricks
Chain or Stacking Exceptions

`ChainedException directory`

What should be done in an exception handler?

Obviously, depends on the application. Here are some possibilities...

- Sometimes, nothing
- A dialog box (warning/error)
- Write a message to a log that an operation failed
- Invoke method System.selfDestruct()

Review
Usage of exceptions

- Exceptions are meant for “unexpected error conditions”.
- Do not overuse
  - Do not put a try around every method call
  - Group related method calls within a try
  - Exceptions incur overhead in the program
- The point is not to abuse exceptions as a way to report expected situations

Review (cont)

- 4 reasons exceptions are thrown
  - Programmer error
    - Ex, Array out of bounds
  - Internal error in Java
  - You call a method that throws an exception
  - You throw an exception

Review Exception Keywords

- *try*
- *catch*
- *finally*
- *throws*
- *throw*