Exceptions

CSE 142, Summer 2002

Computer Programming 1

http://www.cs.washington.edu/education/courses/142/02su/

Readings and References

- Reading
  - » Chapter 18, *An Introduction to Programming and Object Oriented Design using Java*, by Niño and Hosch
  - » Chapter 21, *Introduction to Programming in Java*, Dugan
- Other References
  - » "Handling Errors with Exceptions", Java tutorial
  - » http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html

Exceptions

- In the old days, error conditions were usually handled with returned error codes
- If function really returns a value, can have problems distinguishing between error condition and legitimate value
  - » which value will never, ever occur normally?
- Exceptions provide a much more elegant way to deal with error conditions!

Exceptional Conditions

- An “exceptional condition” is a problem that prevents the continuation of the method or scope that you’re in
- You can’t continue because the current context doesn’t have enough *information* to solve the problem then and there
- There is no definite rule about what’s an exceptional condition and what is not. Use common sense!

Define common sense: return value, ...

- If the problem is something that might happen quite often in perfectly good code, then a distinguished return value might suffice
  - » BufferedReader **`readLine()`** returns a String containing the contents of the line **or** returns null if the end of the stream has been reached
  - » ArrayList **`indexOf(Object elem)`** returns the index of the first occurrence of the argument in the list **or** returns -1 if the object is not found

…, checked exception, ...

- If the problem is something that might happen from time to time (but not often) in well written code, you can use a checked exception
  - » the compiler checks that all checked exceptions are caught by error handling code somewhere
  - » **`FileReader(File f)`** throws FileNotFoundException if the specified file is not found
  - » **`InetAddress getByName(String host)`** throws UnknownHostException if no IP address for the host could be found

…, runtime exception, ...

- If the problem will not happen in well written code, then you don't need to check for it
  - » subclasses of RuntimeException
  - » **`ArithmeticException`** (eg, divide by zero)
  - » **`NullPointerException`** (eg, try to access an object method with a null reference)
  - » **`IndexOutOfBoundsException`** (eg, attempt to access an array element with an index that is too large or too small)

…, fatal error.

- Some problems are so bad that the program just has to give up entirely
  - » **`NoClassDefFoundError`** - Thrown when JVM tries to load in the definition of a class but no definition of the class can be found
  - » **`NoSuchMethodError`** - Thrown if an application tries to call a specified method of a class and that class no longer has a definition of that method. This error can only occur at run time if the definition of a class has incompatibly changed.

So, common sense is:

- return value
  - » you expect this event to occur
- checked exception
  - » it might occur and you know what to do
- runtime exception
  - » it might occur, but it's a bug, so the program should exit as gracefully as it can
- error
  - » we are in deep trouble, and the JVM should terminate the program immediately

When to Use Exceptions

- Exceptions are meant for "unexpected error conditions".
- Exceptions are not meant for "simple, expected situations"
- Deciding which situations are expected and which are not is a fuzzy area. If the user is involved, expect the unexpected...
- Do not use exceptions as a way to report expected situations. They cost more in execution time than a simple if (condition) check.

Generating Exceptions

- All exceptions are generated by constructors and methods
- A constructor or method declares in its signature which checked exceptions it might "throw"

```
public BusReader(String host,int port) throws IOException {
    Socket s = new Socket(host,port);
    InputStream in = s.getInputStream();
    textReader = new BufferedReader(new InputStreamReader(in));
}
```

- Any method might throw a RuntimeException or Error, regardless of its signature.

from BusReader.java

Handling Exceptions

- 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.

```
try {
… risky business …
}
catch (IOException e) {
… fix it up …
}
```

Example

```
try {
    if (arg.length == 2 && arg[0].toLowerCase().equals("file")) {
        source = new BusReader(arg[1]);
    } else if (arg.length == 3 && arg[0].toLowerCase().equals("net")) {
        int port = Integer.parseInt(arg[2]);
        source = new BusReader(arg[1],port);
    } else {
        System.out.println(usage);
    }
}
catch (IOException e) {
    System.out.println(e);
}
catch (NumberFormatException e) {
    System.out.println(e);
}
```

from BusDisplay.java

`finally` block

- What if there is some code we want to execute whether or not there are problems?
- Use a **`finally`** block after the last **`catch`** block

```
try {
… risky business …
}
catch (IOException e) {
… fix it up …
}
finally {
… always do this …
}
```


We've caught it, now what?

- What should be done in an exception handler?
  - » Unwind any partially executed state
  - » eg, if first of two files is open but second failed, close the first one
  - » eg, if an ArrayList is allocated, but the mandatory first entry failed, set the list reference to null
- Details depend on the application.
  - » A dialog box (warning/error)
  - » Write a message to a log that an operation failed
  - » Invoke method System.exit()
  - » Sometimes (rarely), do nothing

Handling Exceptions

- If it is not appropriate for your method to handle an exception, simply let it flow upward by adding a **`throws`** to your method's signature

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

- At some point above you in the calling sequence, the exception will be caught

What if several Exceptions are possible?

- 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 use 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 or specify that you might throw the exception yourself, you will get a compile error.

Catching a superclass is okay

- 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, for example
  - » if FileNotFoundException extends IOException
  - » and FileNotFoundException is thrown
  - » it is permissible to catch either FileNotFoundException or IOException.
- It is usually advisable to be as specific as you can in what you catch. This encourages more appropriate error handling.

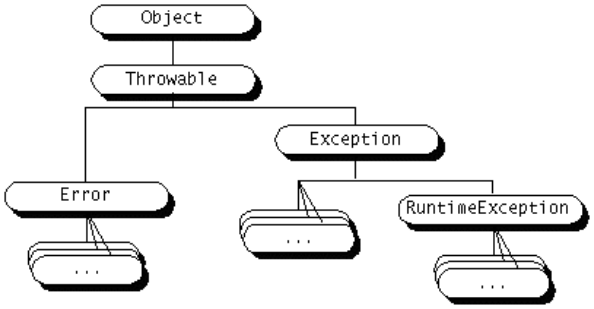
Try / Catch Summary

- Execution enters a try{} block
- If an exception is thrown, execution jumps immediately to the appropriate catch{} block
- If an exception is not thrown, execution proceeds through the try{} block, then skips all the catch {} blocks
- In either case (exception or not), when execution exits the try{} block, it proceeds to a finally{} block if there is one.

ExceptionExample.java

Exceptions are objects

- Exceptions in Java are (not surprisingly) defined in classes
- All such classes inherit from java.lang.Throwable



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 catch RuntimeExceptions, but not generally a good idea. Fix the bug instead!

java.lang.Error

- Reserved for fundamental, usually serious problems at the VM level
- Should not be caught--they can occur anywhere, so trying to handle them is impractical.
  - » Usually they are fatal anyway
- For example: OutOfMemoryError, ClassFormatError, NoClassDefFoundError
- It is very rare to define new Error subclasses

java.lang.Throwable

- In a `catch(Exception e){}`, 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

Throwing Exceptions

- 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.

```
if (bad_news) {
   throw new FrammisException("Unexpected frammis index: "+i);
} else {
   // continue with business of your method
}
```

Defining New Exceptions

- 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.
- By convention, class Throwable and its subclasses have two constructors
  - » one that takes no arguments
  - » one that takes a String argument that can be used to produce an error message.

Defining new Exception classes

```
public class FrammisException extends Exception {
    public FrammisException() {
        super();
    }
    public FrammisException(String s) {
        super(s);
    }
}
```


Naming the new Exception

- Try to give Exceptions names that make sense. Think of situations in which the following Exceptions might be thrown:
  - » FileNotFoundException
  - » MalformedURLException
  - » StringIndexOutOfBoundsException
  - » NegativeArraySizeException