# CS 61B Data Structures and Programming Methodology

July 7, 2008

David Sun

# Announcements

- You’ve started (or finished) project 1, right?

# Package Visibility

- **public** declarations represent specifications—what clients of a package are supposed to rely on.
- package private declarations are part of the implementation of a class that must be known to other classes that assist in the implementation.
- **protected** declarations are part of the implementation that subtypes may need, but that clients of the subtypes generally won't.
- **private** declarations are part of the implementation of a class that only that class needs.

| | Same Package | Subclass | Everywhere |
|---|---|---|---|
| public | Yes | Yes | Yes |
| protected | Yes | Yes | No |
| default | Yes | No | No |
| private | No | No | No |

# Quiz

```
package SomePack;
public class A1 {
    void f1() {
      A1 a = ...
      a.x1 = 3; // OK
    }
    protected int y1;
    private int x1;
}
```

```
//default package
class A2 {
    void g (SomePack.A1 x) {
        x.f1 (); // OK?        ERROR
        x.y1 = 3; // OK?       ERROR
    }
}
class B2 extends SomePack.A1 {
    void h (SomePack.A1 x) {
        x.f1 (); //OK?         ERROR
        x.y1 = 3; //OK?        ERROR
        f1(); // OK?           ERROR
        y1 = 3; // OK?         OK
        x1 = 3; // OK?         ERROR
    }

    void f1() { //does this    YES
                                //override?
    }
}
```

# Today

- Exception Handling

# In a Perfect World …

- Users would never enter a data of the wrong format.
- Files they want to open always exist.
- And code would never have bugs.

# The Bad News…

- Exceptional cases will arise, and your will need to handle them.
- Why?
  - If a user loses all the work she did during a session because of a programming mistake, that user may forever stay away from your program!
- Professional quality/industrial strength programs devote a large amount of code to handle errors.
- Safety/mission critical systems depend on the code to do something reasonable in the face of exceptions...

A problem has been detected and windows has been shut down to prevent damage
to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.

If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBFE7617,0x00000000)

***  SPCMDCON.SYS - Address FBFE7617 base at FBFE5000, DateStamp 3d6dd67c

# Dealing with Exceptions

- When an exception occurs, the program ought to either
  - Return to a safe state and allow the user to execute other commands
  - Allow the user to save his/her work and gracefully exit the program.
- This is not always easy.
  - Exception detection may be needed at multiple places in your code.
  - The code that restores the state of the application can be far removed from where exceptions can occur.

# Types of Exceptions

- External to the program
  - User Input Errors
    - Typos
    - Invalid format: expect a number when text characters are supplied.
  - Device Errors
    - Hardware don’t always what you want it to.
    - Printer may be turned off or run out of paper in the middle of a printout.
    - Connection to a remote server may die while your application is receiving data.
  - Physical Limitations
    - Disk running out of space.
    - Run out of available memory.

# Types of Exceptions

- Internal Exceptions:
  - Errors in your code.
  - Accessing an invalid array index.
  - Trying to find an nonexistent entry in a table.
  - Using other methods incorrectly.

# Traditionally…

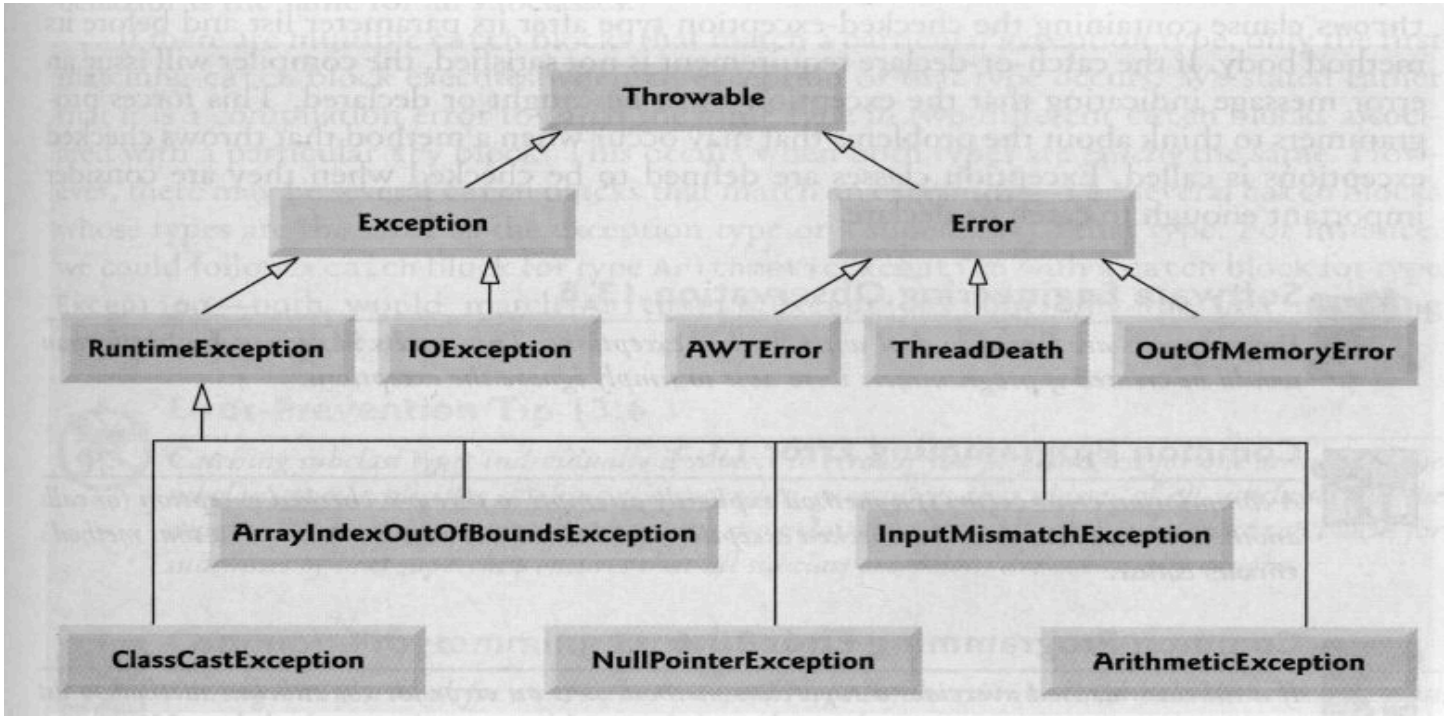
- If a method detected an exception, e.g., an invalid input, the method returns a special error code that the calling method analyzes.
- Example:
  - A file open method return a -1 if the file cannot be found.
  - Reading the end of a file returns -1 end-of-file value marker rather than a standard character.
- What's the drawback?

# Exception Handling in Java

- Java allows every method an *alternative* exit path if it is unable to complete its task in the normal way.
- The method ***throws*** an object that encapsulates the error.
- The method exits immediately, no value is returned.
- The exception handling mechanism looks for an ***exception handler*** that can deal with the error condition.

# Exceptions

- Exceptions are objects.
- All exceptions extend from the `Throwable` class.



# Pre-declared Subtypes

- `Error`:
  - Internal errors and resource exhaustion inside the runtime environment
  - Usually unrecoverable.
- `Exception`:
  - Intended for all other cases.
  - Exceptions that derive from `RuntimeException`:
    - usually because you made a programming error
  - Others that do not derive from `RuntimeException`:
    - bad things that happen to a good program, e.g. `IOException`

# Unchecked Exception

- Intended for programmer Errors
  - If it is a `RuntimeExceptions` it was your fault!
- Examples:
  - Executing `(String) x` when `x` turns out not to point to a `String`: `ClassCastException`
  - Executing `A[i]` when `i` is out of bounds: `ArrayIndexOutOfBoundsException`
  - Executing `x.y` when `x` is null: `NullPointerException`

# Checked Exceptions

- Intended to indicate exceptional circumstances that are not necessarily programmer errors.
- Examples:
  - Attempting to open a file that does not exist: `FileNotFoundException`
  - Input or output errors on a file: `EOFException`
  - Receiving an interrupt: `InterruptedException`

# Throwing Checked Exceptions

- One way to handle exceptions is to not handle it -- pass it on and let the user of the code handle it.
- In this case you advertise that the method can **throw** an exception.

```
public FileInputStream(String name) throws FileNotFoundException
```

- In this example, the constructor can initialize a new `FileInputStream` object *or* it can throw an exception if the file specified by the String is not found.

```
public int read() throws IOException
```

- In this example, the method can return a byte of data from a `FileInputStream` *or* it can throw an exception if some Input-Output (IO) error occurs.

# Example

```
class MyReader{
   . . .
   public void readSolution() throws FileNotFoundException,
   IOException
   {
        f = new FileInputStream("~cs61b/proj1.solution");
        i = f.read();
   }
}
```

# When Exceptions are Thrown

- In any of the four situations:
  1. You called a method that throws a checked exception.
  2. You detected an error and throw a checked exception.
  3. An internal error occurs.
  4. You made a programming error, causing an unchecked exception

# When to Declare Exceptions

- Cases 1 and 2 are checked exceptions
  - You need to tell the programmers who will use your method about the *possibility* of an exception:
  - If the exception is not dealt with, the program terminates.
- Case 3 are errors
  - You do not need to advertise internal Java errors – exceptions derived from `Error`.
  - Any code can throw those exceptions; they are beyond your control.
- Case 4 are unchecked exceptions
  - You should not advertise unchecked exceptions – those that inherit from `RunTimeException`.
  - These are completely under your control

```
//bad style
public void myMethod(int i) throws ArrayIndexOutOfBoundsException
```

# Throwing Exceptions

- If you detected something that's potentially problematic in your code, you can throw an exception.
- Example:
  - You have a method that reads a 16 digit credit card numbers. But you get a 12 digit number from the input file. You decide the situation is so abnormal that you want to through an exception.

```
String readBankAcctNumber(Reader r) throws DataFormatException
{
   while (r.hasNext()) {
        s = r.next();
        if (numberOfDigits(s) != 16)
                throw new DataFormatException();

        . . .
   }
   return s;
}
```

# Creating Your Own Exceptions

- If one of the existing exception classes works for you, then just instantiate an exception object and throw it.
- Otherwise, you can create your own:

```
class AccountNumberFormatException extends DataFormatException {
   public AccountNumberFormatException (){}
   public AccountNumberFormatException (String error) {
        super(error);
   }
}
```

*A convention to have two constructors: One takes no parameters, and one takes an error message in the form of a String.*

```
if (numberOfDigits(s) != 16)
   throw new AccountNumberFormatException ("ERROR: Account number
   not                                     16-digits");
```

# Catching Exceptions

- When an exception is thrown by a method, the calling method can either catch the exception or keep propagating it (by throwing).
- If the exception doesn't get handled by anyone, Java will try to handle it: terminate your program and print a stack trace.
- To catch an exception, set up a try/catch block

```
try {
  Stuff that might throw exception;
  Other code;
} catch (SomeException e) {
  Handler: do something reasonable;
}
Go on with life;
```

# Example

```
class MyReader{
   . . .
   public void readSolution() throws FileNotFoundException,
   IOException
   {
        f = new FileInputStream("~cs61b/proj1.solution");
        i = f.read();
   }
}

class MyReader{
   . . .

   public void readSolution()
        try {
          f = new FileInputStream("~cs61b/proj1.solution ");
          i = f.read();
        } catch (FileNotFoundException e1) {
          whine("Foiled!!");
        } catch (IOException e2) {
          f.close();
        }
   }
}
```

# Scoping Rules

```
class MyReader{
   . . .

   public void readSolution()
        try {
          FileInputStream f = new            FileInputStream
          ("~cs61b/proj1.solution ");
          i = f.read();
        } catch (FileNotFoundException e1) {
          whine("Foiled!!");
        } catch (IOException e2) {
          f.close();
        }
   }
}
```

f *is only visible in the try block.*

# Catching Multiple Exceptions

- You can catch multiple exception types:

```
try {
  Stuff that might throw exception;
  Othercode;
} catch (ExceptionType1 e) {
  Handler: do something reasonable;
} catch (ExceptionType2 e2) {
  Handler: do something reasonable;
} catch (ExceptionType3 e3) {
  Handler: do something reasonable;
}
Go on with life;
```

- The order in which the exception are processed matters:
  - `ExceptionType1` needs to be *less general* than `ExceptionType2`.
  - You need to catch a `FileNotFoundException` object before catching an `IOException` object and before you catch a `Exception` object.
- You can catch the most general exception with

  `catch (Exception e)`

  But you have less control on how to process the Exception

# finally

- When the code throws an exception, execution transfers to the exception handler or if no exception handler is found the method exits.
- What if you want to clear up resources?
  - You put all the cleaning up code in the `finally` clause:

```
try {
  Stuff that might throw exception;
  Othercode;
} catch (SomeException e) {
  Handler: do something reasonable;
} catch (SomeOtherException e2) {
  Handler: do something reasonable;
} catch (SomeMoreException e3) {
  Handler: do something reasonable;
} finally {
  Release all resources, e.g. close files and network
  connections
}
Go on with life;
```

# Next Time

- Reading
  - Head First Java, pp. 315-338
- Next Time:
  - Inner classes
  - Object Cloning
  - Loose Ends