CSCI 112
Wednesday, July 14, 2010

Exception Handling

- handling the program when things go wrong.

Sometimes errors are avoidable, you can write things so that things just won't happen. Other times things are unavoidable, and the user is able to type things in that go wrong.

What can go wrong?

- **division by zero:** any number divided by zero will abort your program.

```
4 / 0
```

- **Array index out of bounds:** try to access an array element out of the array, your program will abort

```
a[-1]
```

- **Accessing files which cannot be found:** if a file can't be found, it will halt your program.

```
new File( “does_not_exist.txt” );
```

- **Following null references:** References point to objects, and when they don't point to anything, they point to “null” this is known as a “null pointer exception.

```
Scanner scan = null;
scan.nextInt();
```

- **Reading an integer using characters:**

```
Integer.parseInt( “Apple” );
```

Three Ways to Handle Exceptions

1. Do nothing, maybe the problem will resolve itself. “When a Java program encounters an exception that isn't handled, it will abort the program and print a “Stack Trace” to the screen. This is handy if you are debugging, because it will report all of the method calls from the start of the program to the current method and which line number called which method.”
2. Pass the problem to another method.
3. Handle the problem right now.
    - Fixing the issue (assuming you know how)
    - Abort the program and explain why the issue halts the program.

Pass the Problem to Another Method

Passing a problem to the caller method is the only way to pass a problem to another method. This is known as **throwing** the exception. You have to specify in the method name that there could be a problem and this method will throw exceptions.

```
void myMethod() throws Exception{
	//statements
	if(/*test for bad conditions*/){
		throw new Exception( "Bad Condition Happened." );
	}
}
```

Handle the Problem Right Now

```
void myMethod(){
	try{
		//statements
		if( /*test for bad condition*/ ){
			throw new Exception( "Bad condition happened." );
		}
	}
	catch( Exception e ){
		System.out.println( "Caught the exception" + e );
		System.exit(1);
	}
}
```

How to catch a problem in Scanner's "nextInt()" method:

```
Scanner scan = new Scanner( System.in );

int n = 0;

while(true) {
	try{
		n = scan.nextInt();
		break;
	}
	catch( InputMismatchException e ){
		System.out.println( "Didn't type a number." );
	}
}
System.out.println( "You typed number " + n );
```

To better illustrate the exception handling traits in Java, this good example is found in the book:

```
// From "Java Program" by Malik, 1-4188-3540-4
// Chapter 12, Page 773
//

import java.io.*;

public class PrintStackTraceExample1 {
   public static void main(String[] args) {
      try {
         methodA();
      }
      catch (Exception e) {
         System.out.println(e.toString() + " caught in main");
         e.printStackTrace();
      }
   }

   public static void methodA() throws Exception { methodB(); }

   public static void methodB() throws Exception { methodC(); }

   public static void methodC() throws Exception {
      throw new Exception("Exception generaged in method C");
   }
}
```

Checked vs. Unchecked Exceptions

Checked Exception – requires that it must be caught of thrown.

- include exceptions related to file I/O

Unchecked Exception – may be ignored

- Classes that extend from RuntimeException are unchecked.

- All other exceptions are checked.

How to Check if an Exception is Checked or Unchecked

Making your own exceptions simply requires making a new class and extending Exception. Make the class name the name of the exception and make the constructor accept a String parameter and then call "super" on that parameter.

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

Using your exception class like you would any other exception error.

```
void myMethodWhichDoesStuff() throws MyOwnException {
    // Statements
    if ( /* bad condition */ )
        throw new MyOwnError("My Own Error Exception");
}
```

IO Exceptions

The System object.

Three static fields in System:

- they are public
- they are final

1. in → InputStream, reads from keyboard.
2. out → PrintStream, prints to the screen.
3. err → PrintStream, prints to the screen (err is short for error).

```
class PrintTest{
        public void static main( String[] args ){
                System.out.println( "Hello." );
                System.err.println( "There is a problem." );
        }
}
```

If you were to execute this you would see printed to the screen:

```
Hello.
There was a problem.
```

It is possible to run it in the command line without printing the error messages to the screen.

Writing To a File

- causes a checked exception

```
import java.io.*;

class FilewriterTest{
	public static void main( String [] args ) throws IOException{
		String filename = "data.txt";
		FileWriter fw = new FileWriter( finename );
		BufferedWriter bw = new BufferedWriter( fw );
		PrintWriter outfile = new PrinterWriter( bw );

		outfile.println( "Hello, textfile." );

		outfile.close(); //essential to make sure file is properly closed
	}
}
```