# Announcements

- Project 6 now out.
  - » Milestone due Oct. 24th
  - » Final project due Oct. 31
- Exam 2
  - » Wednesday Oct. 31, 7-8 pm
  - » EE 129
- Deadline to drop
  - » Wednesday Oct. 24th

# Some Terminology

- *Throwing an exception*: either Java itself or your code signals when something unusual happens
- *Catching an exception*: responding to an exception by executing a part of the program specifically written for the exception
  - » also called *handling an exception*
- The normal case is handled in a `try` block
- The exceptional case is handled in a `catch` block
- The catch block takes a parameter of type `Exception`
  - » it is called the *`catch`-block parameter*
  - » `e` is a commonly used name for it
- If an exception is *thrown,* execution in the `try` block ends and control passes to the `catch` block(s) after the `try` block

# `try-throw-catch` Program Flow

Try block

Statements execute up to the conditional `throw` statement

If the condition is `true` the exception is thrown

- » control passes to the `catch` block(s) after the `try` block

Else the condition is `false`

- » the exception is not thrown
- » the remaining statements in the `try` block (those following the conditional throw) are executed

Catch block

Executes if an exception is thrown

- » may terminate execution with `exit` statement
- » if it does not exit, execution resumes after the `catch` block

Statements after the Catch block

Execute if either the exception is not thrown or if it is thrown but the `catch` block does not exit

# An Example of Exception

`ExceptionDemo`
try and catch blocks

Note: You would normally not use `Exception` here, but an exception class you created yourself, such as `MilkException`.

```
try
{
   System.out.println("Enter number of donuts:");
   donutCount = SavitchIn.readLineInt();

   System.out.println("Enter number of glasses of milk:");
   milkCount = SavitchIn.readLineInt();

   if (milkCount < 1)
     throw new Exception("Exception: No Milk!");

   donutsPerGlass = donutCount/(double)milkCount;
   System.out.println(donutCount + " donuts.");
   System.out.println(milkCount + " glasses of milk.");
   System.out.println("You have " + donutsPerGlass
            + " donuts for each glass of milk.");
}

catch(Exception e)
{
   System.out.println(e.getMessage());
   System.out.println("Go buy some milk.");
   System.out.println("Program aborted.");
   System.exit(0);
}
```

# Java Predefined Exceptions

- `Exception`
  - » The parent of all other exception classes
  - » If you want to use the same error-handling code for all exceptions, you can just catch `Exception`. (More on this later.)
- `IOException`
  - » Often thrown when errors occur reading input or writing output to the screen, to a file, or other types of I/O.
  - » E.g. reading a file that is not in the correct format.
- `FileNotFoundException`
- These are exceptions that you will want to catch in your code, if you are using methods that may throw these exceptions.

# What about these exceptions...

- `ArrayIndexOutOfBoundsException`
  - » For example, if an array has 10 elements and you try access the 11th element: `myArray[10]`.
- `NullPointerException`
  - » When you try to use an object that is set to null. For example:

```
myObject = null;
myObject.myMethod();  // Exception!
```

- Unlike other exceptions, you should not catch these. Always check the array's length or if the object is equal to null if there is a possibility these exceptions may be thrown.
- Note: These are common exceptions in student projects, which often result in poor grades because your program terminates before we can finish testing it...

# Defining Your Own Exception Classes

```
public class DivideByZeroException extends Exception
{
    public DivideByZeroException()
    {
        super("Dividing by Zero!");
    }
    public DivideByZeroException(String message)
    {
        super(message);
    }
}
```

- Note that this only defines the exception class – you must still include code to throw and catch the exception in other classes.
- Often the only methods you need to define are constructors.

# Java Tip: Preserve **getMessage** When You Define Exception Classes

```
throw new Exception("This is a big exception!");
```

This string is stored in an instance variable in the exception object and is returned by the **getMessage** method.

To preserve the correct `getMessage` behavior in `Exception` classes that you define, include:

- a constructor with a string parameter that begins with a call to `super`

```
public DivideByZeroException(String message)
{
    super(message);
}
```

- a default constructor that passes a default message to the `super` constructor

```
public DivideByZeroException()
{
    super("Dividing by Zero!");
}
```

# When to Define Your Own Exception Class

- When you use a `throw` statement in your code, you should usually define your own exception class.
- If you use a predefined, more general exception class, then your `catch`-block will have to be general.
- A general `catch`-block could also catch exceptions that should be handled somewhere else.
- A specific `catch`-block for your own exception class will catch the exceptions it should and pass others on.

# Example: Using the **Divide-ByZero-Exception**

```
public void doIt()
{
    try
    {
        System.out.println("Enter numerator: ");
        int numerator = SavitchIn.readLineInt();
        System.out.println("Enter denominator: "0;
        int denominator = SavitchIn.readLineInt();
        if (denominator == 0)
            throw new DivideByZeroException();
        double quotient =
            (double)numerator/(double)denominator;
        System.out.println(numerator + "/" +
            + denominator + "=" + quotient);
    }
    catch (DivideByZeroException e)
    {
        System.out.println(e.getMessage());
        secondChance();
    }
}
```

This could have been another call to `doIt()`. This is a common use of recursion and exceptions.

# Passing the Buck—Declaring Exceptions

When defining a method you must include a `throws`-clause to declare any exception that might be thrown but is not caught in the method.

- Use a *`throws`-clause* to "pass the buck" to whatever method calls it (pass the responsibility for the `catch` block to the method that calls it)
  - » that method can also pass the buck, but eventually some method must catch it

- This tells other methods

  "If you call me, you must handle any exceptions that I throw."

# Example: `throws`-Clause

`DoDivision`

- It may throw a `DivideByZeroException` in the method `normal`
- But the `catch` block is in `main`
- So `normal` must include a *`throws`-clause* in its declaration statement:

```
public void normal() throws DivideByZeroException
{
   <statements to define the normal method>
}
```

# Example: `throws`-Clause

`DoDivision`

- The method `normal`, located in the `DoDivision` class, may throw a `DivideByZeroException`.
- But the `catch` block is in `main`
- So `normal` must include a *`throws`-clause* in its declaration statement:

```
public void normal() throws DivideByZeroException
{
    ...
    if (denominator == 0)
        throw new DivideByZeroException();
    quotient = numerator/(double)denominator;
    ...
}
```

# Example: `throws`-Clause

- In the `main` method:

```
public static void main (String[] args)
{
    DoDivision doIt = new DoDivision();
    try
    {
        doIt.normal();
    }
    catch(DivideByZeroException e)
    {
        System.out.println(e.getMessage());
        // ... other error-handling code
    }
    System.out.println("End of program");
}
```

# More about Passing the Buck

Good programming practice:

Every exception thrown should eventually be caught in some method

- Normally exceptions are either caught in a `catch` block or *deferred* to the calling method in a `throws`-clause
- If a method throws an exception, the catch block must be in that method unless it is deferred by a `throws`-clause
  - » if the calling method also defers with a `throws`-clause, its calling program is expected to have the `catch` block, etc., up the line all the way to `main`, until a `catch` block is found

# Uncaught Exceptions

- In any one method you can catch some exceptions and defer others
- If an exception is not caught in the method that throws it or any of its calling methods, either:
  - » the program ends, or,
  - » in the case of a GUI using Swing, the program may become unstable
- "Exceptions" derived from the classes `Error` and `RunTimeError` do ***not*** need a `catch` block or `throws`-clause
  - » they look like exceptions, but they are not descendants of `Exception`

# Multiple Exceptions and `catch` Blocks in a Method

- Methods can throw more than one exception
- The `catch` blocks immediately following the try block are searched in sequence for one that catches the exception type
  - » the first catch block that handles the exception type is the only one that executes
- Specific exceptions are derived from more general types
  - » both the specific and general types from which they are derived will handle exceptions of the more specific type
- So put the `catch` blocks for the more specific, derived, exceptions early and the more general ones later

**Catch the more specific exception first.**

`MethodA` throws `MyException` but defers catching it (by using a `throws`-clause:

```
public void MethodA() throws MyException
{
   throw new MyException("Bla Bla Bla");
}
```

## Typical Program Organization for Exception Handling in Real Programs

`MethodB`, which calls `MethodA`, catches `MyException` exceptions:

```
public void MethodB()
{
   try
   {
      MethodA();//May throw MyException exception
   }
   catch(MyException e)
   {
      <statements to handle MyException exceptions>
   }
}
```

# Multiple Exceptions and `catch` Blocks in a Method

Suppose we have the following exception classes:

```
class NegativeNumberException extends Exception
    { ... }

class DivideByZeroException extends Exception
    { ... }
```

In addition, in the same class as the `main` method we have the method:

```
double ratio (double numer, double denom) throws
  DivideByZeroException
    { ... }
```

# Multiple Exceptions and `catch` Blocks in a Method

```
public static void main (String[] args)
{
    try
    {
        System.out.println("How many widgets were produced?");
        int widgets = SavitchIn.readLineInt();
        if (widgets < 0)
            throw new NegativeNumberException();
        System.out.println("How many were defective?");
        int defective = SavitchIn.readLineInt();
        if (defective < 0)
            throw new NegativeNumberException("widgets");
        double ratio = findRatio(widgets,defective);
        // the ratio method throws DivideByZeroException
    }
    // catch blocks on next slide....
```

# Multiple Exceptions and `catch` Blocks in a Method

What if `catch(Exception)` had been first?

```
    // main continued
    catch (DivideByZeroException e)
    {
        System.out.println ("Congrats! A perfect record!");
    }
    catch (NegativeNumberException e)
    {
        System.out.println
         ("Cannot have a negative number of " + e.getMessage());
    }
    catch (Exception e)
    {
        System.out.println
           ("Exception thrown: " + e.getMessage());
    }
} // end main
```

# Creating your own Exceptions

- The previous example brings up an interesting point.
- Most of our exception classes have been very simple, with only a constructor or two, and a getMessage() method. You may be asking, “What’s the point of making such a simple class?”
- By creating multiple exception classes, we can check to see what kind of exception was thrown and thus what kind of error happened.
- This allows us to write specific error-handling code as above, since some methods may cause multiple errors that each require different error-handling.

# Exception Handling: Reality Check

- Exception handling can be overdone
  - » use it sparingly and only in certain ways

- If the way an exceptional condition is handled depends on how and where the method is invoked, then it is better to use exception handling and let the programmer handle the exception (by writing the catch block and choosing where to put it)

- Otherwise it is better to avoid throwing exceptions

- An example of this technique is shown in the case study **A Line-Oriented Calculator**

# Case Study: A Line-Oriented Calculator

- Preliminary version with no exception handling written first
- Exception when user enters unknown operator
- Three choices for handling exception:
  - » Catch the exception in the method `evaluate()` (where it is thrown)
  - » Declare `evaluate()` as throwing the exception and catch it in `doCalculation()`
  - » Declare both `evaluate()` and `doCalculation()` as throwing the exception and handle it in `main()`
- Asks user to re-enter the calculation, so it uses the third option
- Also includes an exception for division by zero

# Calculator `evaluate` Method

```
public double evaluate(char op, double n1, double n2)
      throws DivideByZeroException, UnknownOpException
{
 double precision = 0.001;
...
   case '+':
        answer = n1 + n2;
        break;
   ...
   case '/':
        if ((Math.abs(n2) < precision)
              throw new DivideByZeroException();
        answer = n1/n2;
        break;
   default:
        throw new UnknownOpException(op);
...
}
```

*cases for – and * ops not shown here—see text for complete code*

Any number smaller than **precision** is treated as zero because it is usually best to avoid using the equality operator with floating-point numbers.

Throws exception if op is any **char** other than **+**, **-**, *, /

# Calculator **UnknownOpException** Class

```
public class UnknownOpException extends Exception
{
   public UnknownOpException()
   {
       super("UnknownOpException");
   }
   public UnknownOpException(char op)
   {
       super (op + " is an unknown operator.");
   }
   public UnknownOpException(String message)
   {
       super(message);
   }
}
```

Provides an easy way to make the unknown op character part of the exception message.

Note that all three constructors provide a way for the exception object to have a meaningful message.

# Case Study: A Line-Oriented Calculator

- Why did we do the exception handling this way?
  - » We could have caught DivideByZeroException within the evaluate method. (Or have never thrown an exception and put the error-handling code in an if statement.)
    - – But what would we have done? Returned 0? If we printed an error what would the method return? Easiest to let calling function decide.
  - » We could have caught UnknownOpException within the evaluate method.
    - – But again, what would we have done and what would we have returned?
- In summary: Exceptions are very useful if the error handling is dependent upon the context in which the method was called.

# The `finally` Block

At this stage of your programming you may not have much use for the `finally` block, but it is included for completeness - you may have find it useful later

- You can add a `finally` block after the `try`/`catch` blocks
- `finally` blocks execute whether or not `catch` block(s) execute
- Code organization using `finally` block:

```
try block
catch block
finally
{
   <Code to be executed whether or not an exception is thrown>
}
```

# Three Possibilities for a `try-catch-finally` Block

- The `try`-block runs to the end and no exception is thrown.
  - » The `finally`-block runs after the `try`-block.
- An exception is thrown in the `try`-block and caught in the matching `catch`-block.
  - » The `finally`-block runs after the `catch`-block.
- An exception is thrown in the `try`-block and there is no matching `catch`-block.
  - » The `finally`-block is executed before the method ends.
  - » Code that is after the `catch`-blocks but not in a `finally`-block would not be executed in this situation.

# Why Use a `try-catch-finally` Block?

- When you have code that always should be executed regardless of any errors.
- For example, `try-catch` blocks are very common with file I/O (next chapter!). File I/O can throw many exceptions such as `FileNotFoundException`, `IOException`, etc.
  - » You will want to catch all of these exceptions.
  - » Whether or not an exception was thrown, you always want to close the file when you're done.
  - » Close the file in the `finally` clause.

# Summary
## Part 1

- An exception is an object descended from the `Exception` class
- Descendents of the `Error` class act like exceptions but are not
- Exception handling allows you to design code for the normal case separately from that for the exceptional case
- You can use predefined exception classes or define your own
- Exceptions can be thrown by:
  - » certain Java statements
  - » methods from class libraries
  - » explicit use of the `throw` statement
- An exception can be thrown in either
  - » a `try` block, or
  - » a method definition without a `try` block, but in this case the call to the method must be placed inside a `try` block

# Summary
## Part 2

- An exception is caught in a catch block
- When a method might throw an exception but does not have a `catch` block to catch it, usually the exception class must be listed in the `throws`-clause for the method
- A try block may be followed by more than one catch block
  - » more than one catch block may be capable of handling the exception
  - » the first catch block that can handle the exception is the only one that executes
  - » so put the most specific catch blocks first and the most general last
- Every exception class has a `getMessage` method to retrieve a text message description of the exception caught
- Do not overuse exceptions