CSE 11: Lecture 7

- For-loops
- Switches
- Exceptions and exception handling
- Creating and throwing exceptions
- try-catch blocks
- throws-clause declarations
- Exception handling in SavitchIn

(Reading: Savitch, Ch. 3 and 8)

3 kinds of iterative control constructs in Java

- Java has while loops:

```
int i=0, sum=0;
while(i++<100)
   sum += SavitchIn.readLineInt();
```

- And do-while loops:

```
int i=0, sum=0;
do sum += SavitchIn.readLineInt();
while(i++<100);
```

- And also for loops:

```
int sum=0;
for(int i=0; i<100; i++)
   sum += SavitchIn.readLineInt();
```

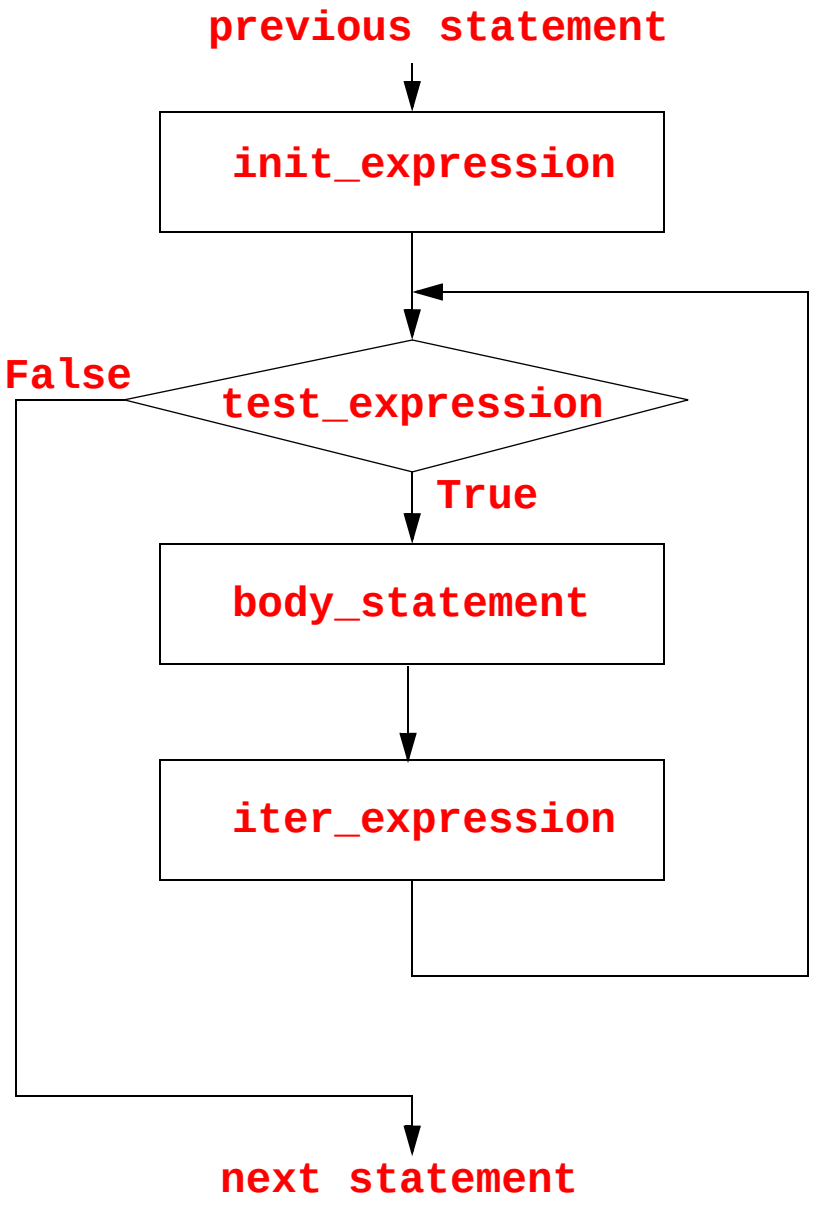
The for statement

- A for statement has the form

```
for (<init_expression>;<test_expression>;<iter_expression>)
  <statement>
```

- When a for statement is executed...
  - **<init_expression>** is executed once.
  - The boolean expression **<test_expression>** is evaluated.
    - If it is “true”, the loop body **<statement>** is executed;
      and then the **<iter_expression>** is evaluated;
      and then the for statement is executed again, starting with **<test_expression>**
    - If it is “false”, execution passes to the statement after the **for** statement
- The **<init_expression>**,**<test_expression>**,**<iter_expression>** are optional: any or all of them can be left out. If the **<test_expression>** is left out, it is taken to be **true**

for-loop flowchart



Declarations in for-loops

- Note that the initialization expression in a for-loop can be a declaration-with-initialization
- This can be very convenient, but watch out for the scope rule:
  - in Java, a variable declared in a for-loop initialization expression is visible within the for-loop *only*

```
// print integers from 0 up to but not including howmany
void printem(int howmany)
{

   for(int i=0; i<howmany; i++)  {
      System.out.println(i);          // okay, in scope of i
   }

   System.out.println( "total: " + i); // error! i out of scope


}
```

Equivalence of iterative control constructs

- As you know, anything you can do with a do-while loop, you can also do with a while-loop, and vice-versa (you may have to add, remove, or change some statements)

- Also: anything you can do with a for-loop, you can also do with a while-loop, and vice-versa (by adding, removing and changing some statements)

```
while(<exp>) <body>
  equivalent to
for(;<exp>;) <body>

for(<init>;<exp>;<incr>) <body>
   equivalent to
{ <init>; while(<exp>) { <body> <incr>; } }
```

- If you have one of these constructs, having the other two does not increase the number of different functions you can compute...

  but it can increase convenience of writing code, and understandability in reading it

Doing the same thing 3 ways...

```
int sum=0;
for(int i=0; i<100; i++){
   System.out.println("Enter a number: ");
   sum += SavitchIn.readLineInt();
}
```

```
int sum=0, i=0;
while(i<100) {
   System.out.println("Enter a number: ");
   sum += SavitchIn.readLineInt();
   i++;
}
```

```
int sum=0, i=0;
do {
   System.out.println("Enter a number: ");
   sum += SavitchIn.readLineInt();
   i++;
} while(i<100);
```

Style suggestions for writing loops

- ✔ It is possible to use **`while`**, **`do-while`**, and **`for`** statements to do the same thing, but:

  - ✗ **`for`** statements are best for loops that are performed a prescribed number of times

    - initialize, check, and increment a control variable that "counts" how many times you've been through the loop

  - ✗ **`while`** and **`do-while`** statements are best for loops that are controlled by some logical condition (boolean expression)

    - If the loop body must be executed at least once, use a **`do-while`** statement

    - If it possible that the loop body will not be executed even once, use a **`while`** statement

Practice with loops

```
int n=50;

while (n>0) System.out.println(n--);
```

How many lines printed?_____ Largest number printed?_____

```
int j = 1;
do {
  System.out.println(j);
  j += 2;
} while (j < 50);
```

How many lines printed?_____ Largest number printed?_____

```
for (int i=0; i<5; i++)
        for (int j=0; j<10; j++)
            for (int k=0; k<20; k++)
               System.out.println(k);
```

How many lines printed?_____ Largest number printed?_____

3 kinds of branching control construct in Java

- Java has if statements:

```
if(a.equals(b))
   operate_on(a,b);
```

- And if-else statements:

```
if(a.equals(b))
   operate_on(a,b);
else operate_on(b,a);
```

- And also switch statements...

Selecting among alternatives

- To perform different actions depending on some alternative values of an expression, one could write a hairy nested “if-else”...

```
if(input == 'A') do_one_thing();
  else if(input == 'B') do_another_thing();
    else if(input == 'C') do_yet_another_thing();
```

...etc.

- Java provides *switch statements* as a “cleaner”, more elegant solution to this type of problem

Switch statements: syntax

- A switch statement has a form like

```
switch (<expression>)
 {
    case <constant1>:
     <statement> .....
      break;
    case <constant2>:
      <statement>
      <statement>
      <statement> ...
      break;
....
    case <constantN>:
      <statement>
      <statement> ...
      break;
    default:
      <statement>
      <statement> ...
 }
```

Switch statements: semantics

- When a switch statement is executed
  - the controlling **<expression>** is evaluated
  - that value is compared to the constant expression in each **case** label in the switch body
  - if the value of **<expression>** is equal to the constant, the statements following that case label, up to the next **break** statement or the end of the switch body, are executed (it is illegal to have two case labels with the same constant)
  - if the value of **<expression>** is equal to none of the constants, the statements following the **default** label are executed, if there is a default label; otherwise no statements in the switch body are exectuted. (A default label is optional, but it is usually a good idea to include one)
  - unfortunately, the case label constants can be only be constant expressions of byte, short, int, long, or char types (no variables and no floating or class types)

Switch statements: an example

- Problem : Do one action if the user enters upper- or lower-case A; do another action if user enters upper- or lower-case Q; otherwise do a third action
- Neatly solved using a switch:

```
char letter = SavitchIn.readLineNonwhiteChar();

switch(letter)
   {
      case 'a':
      case 'A':
         System.out.println("Some kind of a");
         break;
      case 'q':
      case 'Q':
         System.out.println("Some kind of q");
         break;
      default:
         System.out.println("Something else");
    }
```

Ways to end a loop

- A loop will terminate when its controlling boolean expression is evaluated and found to be false...
  this is the “normal” way to end a loop

- However, there are other ways to terminate execution of a loop, and these can sometimes be useful:

  - When a **`return`** statement is executed within a method, you immediately return from that method, even if you are in the body of a loop

  - Calling the static method **`exit`** of the **`System`** class immediately kills your program (even if you’re in the body of a loop). It requires an int argument, the exit status that will be passed to the operating system. Under UNIX, pass 0 for normal termination, nonzero for an error condition: **`System.exit(0);`**

  - Throwing an exception can exit a loop body. More on that in a moment.

  - Executing a **`break`** statement will exit a loop body. More on that right now...

The break statement

- When a break statement is executed in a loop body, control immediately passes to the statement after the nearest enclosing loop body (you 'break out of' the loop)

```
int i=0, j=0;

while(true) {
   if(i>=3) break;  // break out of the outer loop
   j=0;
   while(true) {
      if(j>=2) break;  // break out of the inner loop
      System.out.println(i + " " + j);
      j++;
   }
   i++;
}
```

- Note: because they add another possible flow of control in your code, break statements can make loops hard to understand! Use **`break`** with caution...
- (The above example would be much clearer if re-written without using **`break`**!)

The continue statement

- When a continue statement is executed in a loop, control immediately passes to the end of the nearest enclosing loop body, and the increment expression and/or boolean controlling expression for the loop is evaluated (you 'continue' the loop)

```
int goodOnes=0;
String val;

while(goodOnes < 100) {
   val = readInput();
   if ( ! isGood(val) ) continue;

   process(val);
   goodOnes++;

}
```

- Like break statements, continue statements can make loops hard to understand! Use `continue` with caution too

Flow of control

- You know about various kinds of “control constructs” in Java that determine the sequence of statements that are executed:
  - ordinary statement sequencing
  - conditionals (if, if-else)
  - iterative constructs (for, while, do-while)
  - multiway branches (switch)
  - break and continue
  - method calls, and method returns

- Now we will look at the exception throw and catch mechanism in Java, which adds an interesting new kind of control flow, used mainly for error handling

The normal and the exceptional

- In the programming we've been doing so far, we've been writing code to handle normal, expected cases, such as...
  - the user types in well-formatted input,
  - values of arguments passed to methods are within the appropriate range,
  - there is data in a file when we try to read from it, etc.

- Now we'll learn about language features Java provides to help you write programs that also deal with exceptional cases, such as...
  - badly formatted input,
  - values out of range,
  - running out of data when reading from a file, and
  - other things that may violate the assumptions that "normal" processing is based on

Murphy's law: if something can go wrong, it will

- It is good practice in any programming language to write code that is as bulletproof as possible:
  - software should not do anything disastrous, no matter how moronic its users are
  - well-written code tries to do something reasonable no matter what happens

- This could be done with the control constructs you already know about:

```
if ( some_error_condition ) {
   // handle the error: print a message, try to recover,
   // return a special value indicating an error,
   // and/or exit the program, as appropriate
} else {
   // do normal processing
}
```

- In Java, you can also use Java's *exception-handling mechanism*, which is more powerful...

  ... This requires learning about *exceptions*, *throwing* exceptions, and *catching* exceptions

Exceptions: throwing and catching

- Exceptions are objects in Java
- Exceptions are used to signal the occurrence of errors or other “exceptional” conditions
- To signal an exceptional condition, an exception object is created and “thrown”
  - You can write code to throw exception objects in your own programs
  - You can extend existing exception classes to create your own kinds of exception
  - Standard Java library methods and the Java runtime environment can also create and throw exceptions (anyone have any NullPointerExceptions thrown at them yet?)

- When an exception object is thrown, normal execution of your program immediately stops, and the Java interpreter looks for an exception handler to “catch” the exception
  - an exception handler is written as a *catch block* of code
  - The catch block code should be written just to deal with the exception... this permits separating “normal” information processing from “exceptional” processing in your program
  - If your program does not catch a thrown exception, the Java runtime environment will catch it, and then will try to kill your program and print an error message with some information about where the exception was thrown from

Creating an Exception object

- If you want to throw an exception in your code, you can either
  - create an instance of an existing exception class (there are many)
  - define your own exception class, create and throw an instance of it

- One easy thing to do is to create and throw an instance of the **`java.lang.Exception`** class

- **`Exception`** has only two constructors and three public instance methods:

```
public class Exception extends Throwable {

   public Exception () //default ctor
   public Exception (String message) //initialize with a message

   public String getMessage() //returns the message
   public String toString() //returns the message + other info

   public void printStackTrace() // prints info about where the
                                 // exception occurred
```

Creating and throwing an Exception

- You create an Exception object just like you create any object: using **`new`**
- Every exception class should have a constructor that lets you specify a String message to be contained in the object... **`getMessage()`** and **`toString()`** use that
- You throw the Exception object using the keyword **`throw`**
- Usually, creating the Exception object and throwing it is done in a single statement:

```
throw new Exception("Houston, we have a problem");
```

- When an exception is thrown, normal program execution immediately stops and the interpreter looks for a catch block that can handle the exception....
  - The search propagates up through enclosing statements within the current method, and then back through the chain of method calls, until such a catch block is found.
  - This catch block code can be far from where the throw statement is in your code!
- We will look at catching exceptions in a moment, but first let's see how to pass them along instead of catching them...

Throwing an exception from a method

- Sometimes you do *not* want to handle an exception in the same method in which the exception is created and thrown
  - This is very common in library functions:
    - they "know enough" to detect when an error condition happens,
    - but they probably do not know enough about how the application programmer wants to handle such a condition...
    - ...so they leave handling the error to the user code that called them
  - You should write your methods this way, if it's appropriate

- If you write a method that contains code that can throw an exception, but you do not catch that kind of exception in the method, you have to declare that your method can throw that kind of an exception
  - This is done by using the **`throws`** keyword in the method header
  - (Subclasses of RuntimeException and subclasses of Error do not have to be declared in this way... more about that later)

- Savitch calls this "passing the buck": the exception will be thrown back to where the method was called from.

Throwing an exception from a method: an example

- The method **`doIt()`** shown here contains exception-generating code
- However it does not catch the exception; instead it "passes the buck" back to where the method was called from (notice the declaration in the method header)

```
public class Example {

    public void doIt() throws Exception {

        // find out what day it is
        int day = Calendar.getInstance().get(Calendar.DAY_OF_WEEK);
        // if it's Sunday, throw an exception!
        if(day == Calendar.SUNDAY) {
              throw new Exception("I don't work on Sundays!");
        }

       // Normal processing code goes here
        // We do not catch the exception in this method!
       System.out.println("Did it");
    }
```

Throwing an exception from a method, cont'd

- Now suppose we try to call doIt() from some other method...

```
public static void main (String [] argv) {
      Example x = new Example();
      x.doIt();
}
```

- ... this will not compile! Error message:

```
Error:  Exception java.lang.Exception must be caught, or it must be
declared in the throws clause of this method.
```

- So, again there are two choices:
  - 1: Pass the buck again! This is easy, just declare the method can throw. For example, the following will compile:

```
public static void main (String [] argv) throws Exception {
      Example x = new Example();
      x.doIt();
}
```

  - 2: Catch the Exception. Let's see how to do that now

Catching an exception: try blocks and catch blocks

- To catch an exception in Java, you have to
  - enclose the potentially exception-generating code in a “try” block
  - follow the “try” block with one or more “catch” blocks that specify what kinds of exceptions they will catch
- a try block is a block (compound statement) beginning with the keyword **`try`**
- a catch block is a block (compound statement) beginning with the keyword **`catch`** and an exception class and parameter in parentheses

```
try {

      // code that can generate an exception of type EClass
      // goes here

} catch (EClass e) {

      // code to handle an exception of class EClass goes here.
      // e points to the EClass exception object that was thrown

}
```

Catching an exception: an example

- The try-block must contain potentially exception generating code:
  - It can contain an explicit **throw**, or it can contain a call to a method that throws an exception back to the point of call ("passing the buck")

```
public class Example {
   public static void main (String [] argv) {
      Example x = new Example();
      try {
         x.doIt();    // this method call can throw an Exception
         System.out.println("Okay, we did it.");
         // etc... normal flow of control

      } catch (Exception e) {
         // if a Exception is thrown in the associated try-block,
         // this catch-block will be executed to handle it:
         System.out.println("Problem: " + e.getMessage());
         System.exit(-1);  // or whatever is appropriate
      }
}
```

Catching an exception: another example

- Here is an example of a try-block that contains an explicit throw:

```
public class Example {
   public static void main (String [] argv) {
      try {
         // Create a Calendar object to find what day it is
         Calendar c = Calendar.getInstance();
         // if it's Sunday, throw an exception
         if(c.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY) {
               throw new Exception("Sorry, it's Sunday");
         }
            ...
            ...   // rest of code for normal processing goes here

      } catch (Exception e) {
         // if a Exception object is thrown in the try-block,
         // this catch-block will be executed to handle it:
         System.out.println("Problem: " + e.getMessage());
         System.exit(-1);  // or whatever is appropriate
      }
```

Multiple catch blocks

- There are many exception classes in the Java standard libraries (and you can define your own exception classes).
- Usually different exception classes correspond to different types of error conditions
- ... and you probably want to handle different error conditions differently
- Java lets you accomplish this by having multiple catch-blocks attached to the same try-block
- Each catch-block should declare a different type of exception, and contain code for handling that type

Multiple catch blocks: an example

```
public static void main(String[] args) {

   try {

      Integer i = Integer.parseInt("xyzzy");
      i = null;
      System.out.println(i.intValue());
      System.out.println("No problem!");

   } catch (NumberFormatException e) {
      System.err.println("Bad format!");  // handle bad format

   } catch (NullPointerException e) {
      System.err.println("Null pointer!"); // handle null pointer

   } catch (Exception e) {  // handle everything else
      System.err.println("Some random exception: " + e);
   }
}
```

Summary: Exception-handling semantics in Java

- When an exception is thrown, execution of your program immediately stops, while the interpreter looks for a catch block to catch it
- If the throw statement is enclosed in a try-block, catch-blocks attached to the try-block are checked, in order. If no catch block matches, the exception "keeps going"!
- If the exception is not caught within a method, it is thrown by the method back to the place where the method was called
- If that call is within a try-block, catch-blocks following the try-block are checked, in order
- If that call is within another method and not caught there, it is thrown by that other method back to the place in your program where it was called...
- ... etc! Eventually, the exception will be caught, or it will be thrown by your main() method
- If it is thrown by your main method, your program will die, with a diagnostic message

- Resuming normal execution:
  - If the exception is caught by a catch-block, the code in the catch-block is executed to handle the exception, and normal execution resumes with the first statement after the catch-block

A look at SavitchIn

- The SavitchIn class methods show examples of using exceptions for error recovery... and also of not using exceptions for error recovery (buck-passing). Let's look at some of the static methods in that class:
- **`public static char readChar()`** Used to read the next character in the input. Calls System.in.read() to get this character as an int. System.in.read() can throw an IOException; there is a try-catch within the readChar() method to deal with this.
- **`public static String readLine()`** Used to read the next line in the input, up to the next carraige return. Nothing in readLine() can throw an exception; so **`try`**, **`catch`**, or **`throws`** do not appear in its definition
- **`public static int readLineInt()`** Used to read an integer literal constant the next character in the input. It uses Integer.valueOf() to parse the line; this method can throw a NumberFormatException, which is caught.
- **`public static int readInt()`** Used to read a whitespace-delimited integer literal constant.It uses Integer.valueOf(); but instead of catching NumberFormatException, this method "passes the buck" and rethrows it
- **`public static char readLineNonwhiteChar()`** Used to read a single nonwhite character on a line. It handles badly formatted input without using exceptions at all...

  ... Note: following examples have had most of the comments removed or changed

SavitchIn.readChar()

```
public class SavitchIn
{
    public static char readChar()
    {
        int charAsInt = -1; //To keep the compiler happy
        try
        {
            charAsInt = System.in.read();
        }
        catch(IOException e)
        {
            System.out.println(e.getMessage());
            System.out.println("Fatal error. Ending Program.");
            System.exit(0);
        }

        return (char)charAsInt;
    }
```

SavitchIn.readLine()

```
public static String readLine()
    {
        char nextChar;
        String result = "";
        boolean done = false;

        while (!done)    {
            nextChar = readChar();
            if (nextChar == '\n') done = true;
            else if (nextChar == '\r') {
                //Do nothing.
                //Next loop iteration will detect '\n'
            }
            else  result = result + nextChar;
        }

        return result;
    }
```

SavitchIn.readLineInt()

```
public static int readLineInt()  {
    String inputString = null;     boolean done = false;
    int number = -9999;//To keep the compiler happy.
    while (! done) {
        try {
          inputString = readLine();
          inputString = inputString.trim();
          number = (Integer.valueOf(inputString).intValue());
          done = true;
        }
        catch (NumberFormatException e)   {
          System.out.println("Your input number is not correct.");
         System.out.println("Your input number must be");
         System.out.println("a whole number written as an");
         System.out.println("ordinary numeral, such as 42");
         System.out.println("Please, try again.");
         System.out.println("Enter a whole number:");
        }
    }
    return number;
```

SavitchIn.readInt()

```
public static int readInt() throws NumberFormatException    {

    String inputString = null;
    // readWord() is a public static method, not shown here.
    // it returns, as a String, the next whitespace delimited
    // sequence of characters in the keyboard input
    inputString = readWord();

    // parse the String, which should contain a well-formatted
   // integer literal constant.  If not, a
   // NumberFormatException is thrown by Integer.valueOf(),
   // and re-thrown by this readInt() method.
    return (Integer.valueOf(inputString).intValue());
}
```

SavitchIn.readLineNonwhiteChar()

```
public static char readLineNonwhiteChar()     {
    boolean done = false;
    String inputString = null;
    char nonWhite = ' ';  //To keep the compiler happy.
    while (! done)     {
        inputString = readLine();
        inputString = inputString.trim();
        if (inputString.length() == 0)     {
          System.out.println("Your input is not correct.");
          System.out.println("Your input must contain at");
           System.out.println("least one nonwhitespace
character.");
          System.out.println("Please, try again.");
          System.out.println("Enter input:");
        } else {
          nonWhite = (inputString.charAt(0));
          done = true;
        }
    }
    return nonWhite;
}
```

Exceptions: the good and the bad

- Using exceptions can be a good way to signal and handle errors in your own code.
- By organizing all your error handling and recovery code within the try/catch structure, you will, hopefully, end up with cleaner code that is easier to understand.
- Note however: the “passing-the-buck” technique of throwing exceptions from methods can mean that the code that handles an exception is very far from the code that originally threw it... This “non-local” flow of control can make your code harder to understand and debug.
- In any case, knowing about exceptions is an essential part of Java programming

The exception to the rule about exceptions...

- There are certain types of exceptions that do not have to be listed in a **`throws`** clause of a method, even if the method throws them:
  - **`Error`**s: these correspond to situations that are not easily predicted or easily handled, such as the system running out of memory. It is often futile to try to recover from these.
  - **`RuntimeException`**s: these correspond to many common run-time problems, such as illegal casts, number misformatting, null pointers, and array index problems.
- The reason for these exceptions-about-exceptions is that Errors and RuntimeExceptions can thrown from such a large number of places that essentially every method would have to declare it throws them, or include code to catch them. (Note: you CAN catch them if you want)
- You really don't have to memorize all of the different kinds of Errors and RunTimeExceptions... if you don't declare an exception in a **`throws`** clause when you need to, the compiler will tell you!

Classes and subclasses

- You can't throw just any kind of object. The following does not compile:

```
throw new String("help!");
```

```
Error:  Can't throw class java.lang.String; it must be a subclass
of class java.lang.Throwable
```

- ... any object thrown must be an instance of a subclass of the class **`Throwable`** (for example, **`Exception`** and **`Error`** are sublclasses of **`Throwable`**, and **`RuntimeException`** is a subclass of **`Exception`**)

- ...any object that is an instance of a class that is a subclass of **`Error`** or **`RuntimeException`** does not have to be caught, or declared in a throws clause

- This is true of standard library classes as well as exception classes you define yourself

- So, how to define subclasses of classes? And how do subclasses fit into the Java type system?

Next time

- ✔ Objects, classes, and inheritance
- ✔ Visibility and inheritance
- ✔ Overriding methods
- ✔ Dynamic method binding and the Java type system
- ✔ The Object class
- ✔ Final classes

(Reading: Savitch, Ch. 7)