CSE 11: Lecture 6

- Mixing data types in numerical expressions and assignment
- Casts
- Operator precedence and associativity
- Increment and decrement operators
- Loop invariants
- Variable lifetime
- Software design in two paradigms
- Software testing: stubs and drivers
- Tracing for debugging

(Reading: Savitch, parts of Ch 2, 3, 5)

Midterm Exam #1 will be next Thurs Oct 16 , during lecture time

Location: >>> here <<<

Closed-book, closed-notes, no calculators. Bring something to write with, and picture ID!

Coverage: Chapters 1-5, Lectures 1-6, Assignments P1-P2

Midterm review: Discussion sections Wed Oct 15

A practice midterm is available online (PDF format)

Lecture notes are available online

Mixing data types in arithmetic expressions

- As you know by now, a value in Java (for example, the value of a variable or a literal constant or an expression) is of a particular type

- The type of a value is important! It determines how the value is interpreted when it is used in a computation

- But if you build an arithmetic expression using operators whose arguments are of different types, what happens?

- Here are the Java rules for type conversion with binary numeric operators and arguments that are of primitive numeric type (including char, but not boolean):
  - If either operand is of type double, the value of the other operand is converted to double and the operation produces a double value.
  - Otherwise, if either operand is of type float, the value of the other operand is converted to float and the operation produces a float value.
  - Otherwise, if either operand is of type long, the value of the other operand is converted to long and the operation produces a long value.
  - Otherwise, the values of both operands are converted to int and the operation produces an int value.

Mixing data types in assignment

- In Java, you can assign a value of a certain type to a variable of that same type (not too surprising!)

```
int i=3, j;
j = i + 2;          // assigning an int value to an int
```

- But you can also assign a value of a certain primitive type to a variable of a different but "larger" type (where there can never be loss of magnitude information):

```
int i=3;     long j;
j = i + 2;         // assigning an int value to a long: okay
```

- In fact, you can assign a value of any type on the following list to a variable of any type that you can get to by following arrows on this list:

```
byte ──► short ──► int ──► long ──► float ──► double
                    ▲
            char ───┘
```

... and Java will automatically do the type conversion ("promotion") for you.

- Also, Java will permit assigning a non-long integral literal or named constant expression to an integral-type variable if the constant expression has a value that is in the range that the variable can hold
- Since passing arguments is like assignment, these rules apply to that as well
- But anything else is a compile-time error!

Mixing data types in assignment: examples

- Question: Are the following statements legal in Java or not?

```
byte a;
byte b = 0;
char c = 'x';
int i = 100;
double d = 3.0;
final short S = 100;
a = S;
a = i;
a = d;
a = b + b;                                  // tricky...
```

Explicit type conversions in Java: casting

- As we have seen, Java does many type conversions for you automatically
- To "manually" convert the value of an expression to a certain datatype, use a *cast*
- A cast of the value of ***`expression`*** to type ***`typename`*** has the form

  `(typename)expression`

  For example, the expression

  `4 / 5`

  has value **`0`** of type int, while the expression

  `4 / (double)5`

  has value **`0.8`** of type double.

  (....   what about the expression **`(double)( 4/5 )`** ?)

- casts have no side effects! (A cast produces a value of the desired type, but does *not* change the type or value of any already existing variable.)

Casting and assignment

- Because of Java's rules for automatic type conversion in assignment statements, the following is illegal -- Java will try to prevent the loss of magnitude information that can happen when you assign a floating-point value to an integer variable:

```
double x = 3.99;
int i;
i = x;                              // error!
System.out.println(i);
```

- However, if you explicitly cast, the program will compile and run -- Java, in effect, will assume you know what you are doing if you explicitly cast:

```
double x = 3.99;
int i;
i = (int) x;
System.out.println(i);           // prints 3
```

- Explicit casts are permitted among all the numeric types (including `char`)
- Casts are not permitted between `boolean` and any other type
- Casts between reference types is a topic we will get to when we discuss inheritance

Operators: computing with data

- An operator operates on data (variables or constants or the results of other operations)
  - An operator always *returns a value* that can be used in further computation
  - An operator may also have "*side effects*" (things it does other than returning a value): it may change the value of variables, print something, etc.

- The pieces of data an operator operates on are its *arguments* or *operands*
  - If an operator takes one argument it's called "unary"; a "binary" operator takes two, a "ternary" operator takes three

- An operator has *precedence level* and an *associativity direction*, which determine how it combines with other operators in an expression
  - Operator precedence and associativity can be overridden by use of parentheses
  - When in doubt, parenthesize!

Precedence and associativity rules

- Operator precedence:
  - In an expression containing more than one operator, high-precedence operators apply before low-precedence ones
- Operator associativity:
  - In an expression with more than one operator of the same precedence...
    - leftmost operator applies first if the operator has left-to-right associativity
    - rightmost operator applies first if the operator has right-to-left associativity
- The precedence and associativity rules in Java often turn out to be the “usual” ones you already know from algebra and logic
- And the use of parentheses to override the precedence rules is also familiar from mathematics

Java operator precedence and associativity

- This table shows the precedence and associativity of the Java operators we will cover in this course (precedence decreases as you go from top to bottom in the table):

<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td><code>-</code> (unary) <code>+</code> (unary) <code>++</code> <code>--</code> <code>!</code> typecasts</td><td>right to left</td></tr>
<tr><td><code>*</code> <code>/</code> <code>%</code></td><td>left to right</td></tr>
<tr><td><code>+</code> <code>-</code></td><td>left to right</td></tr>
<tr><td><code><</code> <code><=</code> <code>></code> <code>>=</code></td><td>left to right</td></tr>
<tr><td><code>==</code> <code>!=</code></td><td>left to right</td></tr>
<tr><td><code>&&</code></td><td>left to right</td></tr>
<tr><td><code>||</code></td><td>left to right</td></tr>
<tr><td><code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code></td><td>right to left</td></tr>
</table>

Increment and decrement operators

- The unary increment and decrement operators **`++`**, **`--`**
  - high precedence
  - the argument must be an "lvalue" (e.g. a variable)

- **`++foo`** returns the value **`foo + 1`**
  It has the side effect of increasing the value of **`foo`** by 1: *pre-increment*

- **`foo++`** returns the (old) value of **`foo`**
  It has the side effect of increasing the value of **`foo`** by 1: *post-increment*

- **`--foo`** returns the value **`foo - 1`**
  It has the side effect of decreasing the value of **`foo`** by 1: *pre-decrement*

- **`foo--`** returns the (old) value of **`foo`**
  It has the side effect of decreasing the value of **`foo`** by 1: *post-decrement*

Practice with increment and decrement operators

```
public static void main(String args[])
{
  int foo = 3;
  System.out.println(foo);        // prints 3
  System.out.println(foo++);      // prints 3
  System.out.println(foo);        // prints 4
  System.out.println(++foo);      // prints 5
  System.out.println(foo);        // prints 5
  System.out.println(--foo);      // prints 4
  System.out.println(foo);        // prints 4
  System.out.println(foo--);      // prints 4
  System.out.println(foo);        // prints 3
}
```

Combined assignment-and-numeric operators

- Java provides operators which combine a numeric operation with assignment. For example:

```
<variable>  +=   <expression>
```

does exactly the same thing as

```
<variable>  =   <variable> + ( <expression> )
```

That is,

```
foo += bar * 50 - baz
```

is a more compact way of writing

```
foo = foo + (bar * 50 - baz)
```

- And likewise for

```
<variable> -= <expression>
<variable> *= <expression>
<variable> /= <expression>
<variable> %= <expression>
```

Doing the same thing 3 different ways

- These expressions return the same values, and have the same side effects:

```
foo = foo + 1

foo += 1

++foo
```

Understanding expressions

- In the context of these declarations:

```
double x1 = 1.0;
double x2 = 5.0;
double x3 = 2.0;
```

- ... what are the values of these expressions? Pay attention to associativity, precedence, and type-conversion rules:

```
x1 + 1.0 * x2

(x1 + 1.0) * x2

x2 / 2  * x3

(x2 / 2) * x3

x1 ++ + 3

x1 = x2 = x3
```

Understanding loops

- Loops are important and powerful control constructs, that can be hard to get right
- To help write loops that do what you want them to do, think about these things:
  - What initial conditions must be satisfied before execution of the loop?
    - These are called “loop preconditions”
    - Statements executed before the loop is reached establish the preconditions
    - You can also state loop preconditions in a comment, for help in thinking about what is going on
  - What conditions must be satisfied before and/or after each execution of the loop body?
    - These are called “loop invariants”
    - The boolean test in a while or do-while loop can express a loop invariant
    - You can state other loop invariants in comments, for help in thinking about what is going on
  - What actions must be performed during each iteration?
    - Statements in the loop body perform these actions (the boolean test expression can perform actions -- side effects -- as well)

Understanding loops, continued

- Figuring out what a loop does can be tricky...!
- It is a very common mistake to be off by one when thinking about the number of times a loop body will execute
  - these are called “off by one errors” (really) or “fencepost errors”

    For this program fragment,

    How many lines are printed out?
    What is the value of the largest number printed?

```
int j = 1;

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

  - Possible answers for # of lines printed:
    24? 25? 26? 27? .... 50? .... something else?

Understanding loops and loop invariants

- Loop invariants are useful in reasoning about what a loop will do
- It can take some thought to come up with good loop invariants... in this example, here are some relevant ones:

```
int j = 1;

   do {
         System.out.println(j);
         j += 2;
      // LOOP INVARIANTS AT THIS POINT:
      // largest_number_printed == j-2
      // 2*number_of_lines_printed == j-1
      // j is an odd number
   } while (j < 50);
```

- From this we can see that
  - the loop ends when `j==51` (the first odd number that makes `j<50` false)
  - ... and so when the loop ends the largest number printed is _____
  - ... and when the loop ends the number of lines printed is _____

More hints for writing loops

- Java has while-loops: the loop continues *while* the boolean condition is true
- Sometimes it is easier to think in terms of until-loops: an until loop continues *until* its boolean condition becomes true
- Java doesn't have until loops, but you can easily have the effect of them:
  - Think clearly about the condition that must end the loop, and make the while loop boolean expression clearly state the logical *negation* of that condition (DeMorgan's laws can sometimes be useful here)
- As a simple example: You know you want to continue a loop until you are done. You can write code with a structure like this:

```
boolean done = false;

while (!done) {
   // stuff
   if (some_hairy_condition) done = true;
   // more stuff
}
```

Yet more hints for writing loops

- Make sure the test expression for a loop will eventually become false: avoid infinite loops!

```
int i=1;
do
{
    System.out.println(i);  // duh
} while (i>0);
```

--or--

```
int i=1;
while (i<50) ;                // subtle
{
    System.out.println(i++);
}
```

- Make sure the loop body contains only necessary statements... They may be executed many times
  - usually avoid declaration statements in the loop body...
    declaration statements create variables, and are “expensive” to execute

Variable visibility and creation

- We have learned about about Java's visibility ('scope') rules for variables (instance, static, parameter, local), and methods (instance and static)

- And you also know when instances of primitive types, pointer variables, and actual objects are created in Java:
  - primitive type values and reference type pointers: when their declaration is executed
    - if static variable: this happens when the class is loaded (usually, when your program starts)
    - if instance variable: this happens when the object is created
    - if formal parameter: this happens when the method is called
    - if local variable: this happens when execution reaches the declaration
  - objects: when `new` is invoked
    - this use of new may be hidden in the body of a method you call
    - Strings are a special case: a String object is created when a String literal constant is evaluated, or the String concatenation operator is invoked

Variable destruction and lifetime

- But when are these instances destroyed (i.e., when is the memory used for them reclaimed)?
- How and when variables are created and destroyed is the issue of variable *lifetime*
- The basic rule for variable destruction in Java is:

  A variable will be destroyed after it can no longer be accessed in your program

- As for other things, this is somewhat different for primitive types vs. objects...

Destruction of primitive type variables, pointers, and objects

- Destruction of primitive type variables and reference type pointers:
    - if static variable: destroyed when the class is unloaded (usually, when your program ends)
    - if instance variable: destroyed when the object is destroyed
    - if formal parameter: destroyed when the method returns
    - if local variable: destroyed when execution exits the block in which it is declared (for example, when the method returns)

- Destruction of objects:
    - destroyed only if the object cannot be referenced, because no pointer in your program is pointing to the object
    - the Java *garbage collector* determines that your program is no longer able to reference the object, and reclaims its memory to use in creating other objects, if needed

- As a result, in Java, you do not have to worry about memory management for variables as much as in some other languages: their lifetime follows a few simple rules, and much of the hard work is done automatically for you

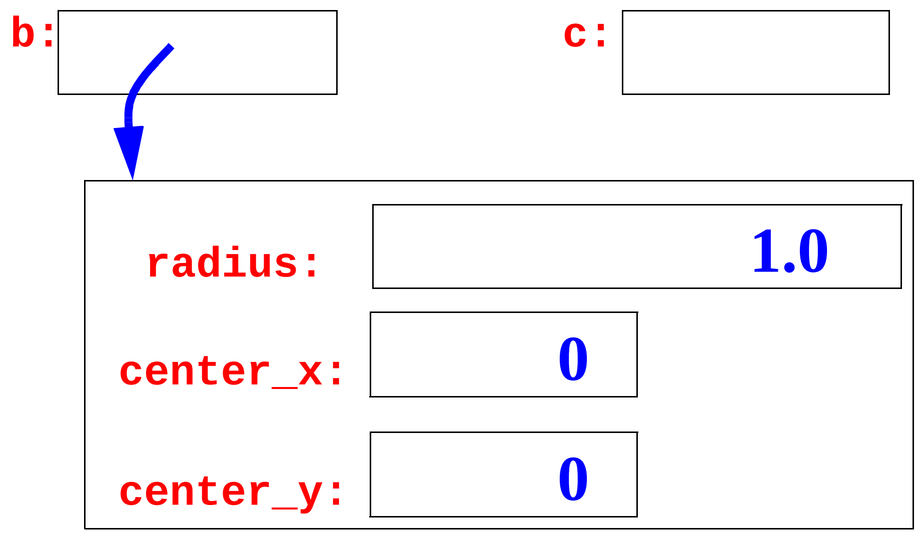
Lifetime of objects: an example, frame 1

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

create new Circle object, make b point to it.
The constructor initializes the instance variables

```
Circle b,c;

b = new Circle();
```



Lifetime of objects: an example, frame 2

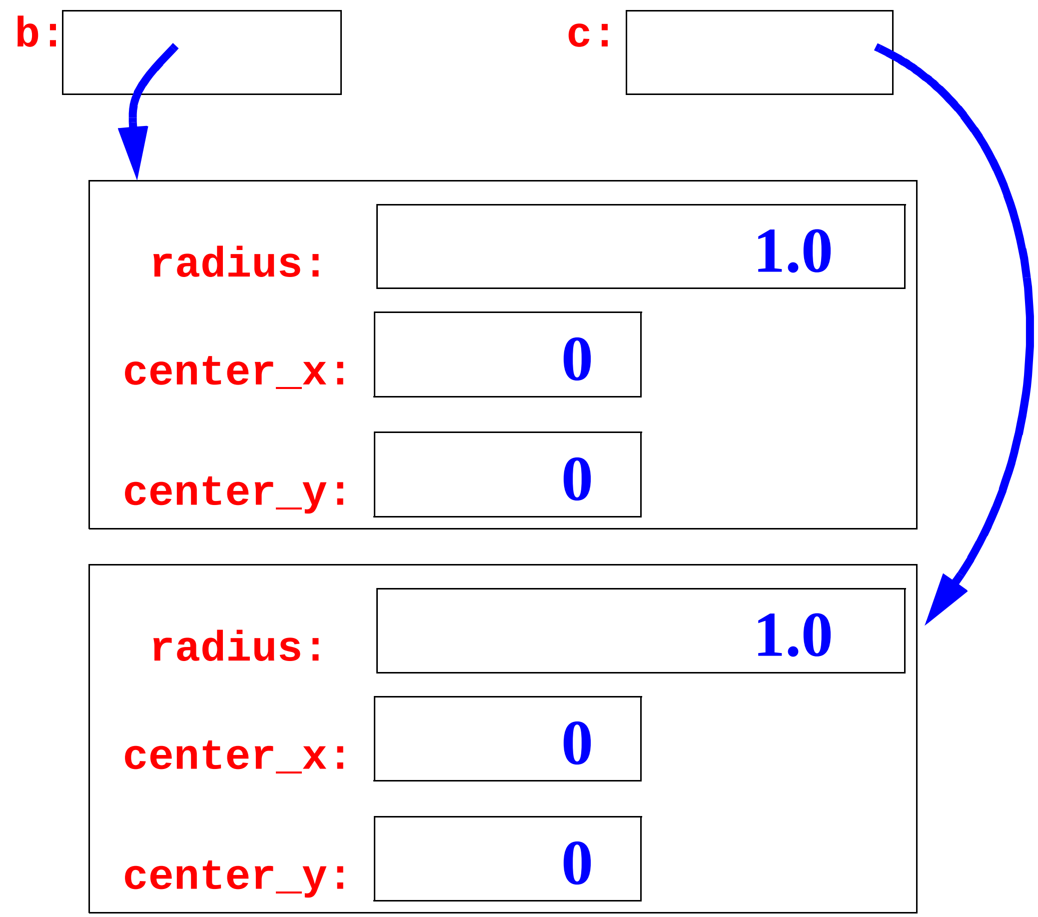
```
public static void main (String args[]) {
```

create another Circle object, make c point to it

```
Circle b,c;

b = new Circle();

c = new Circle();
```



Lifetime of objects: an example, frame 3

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

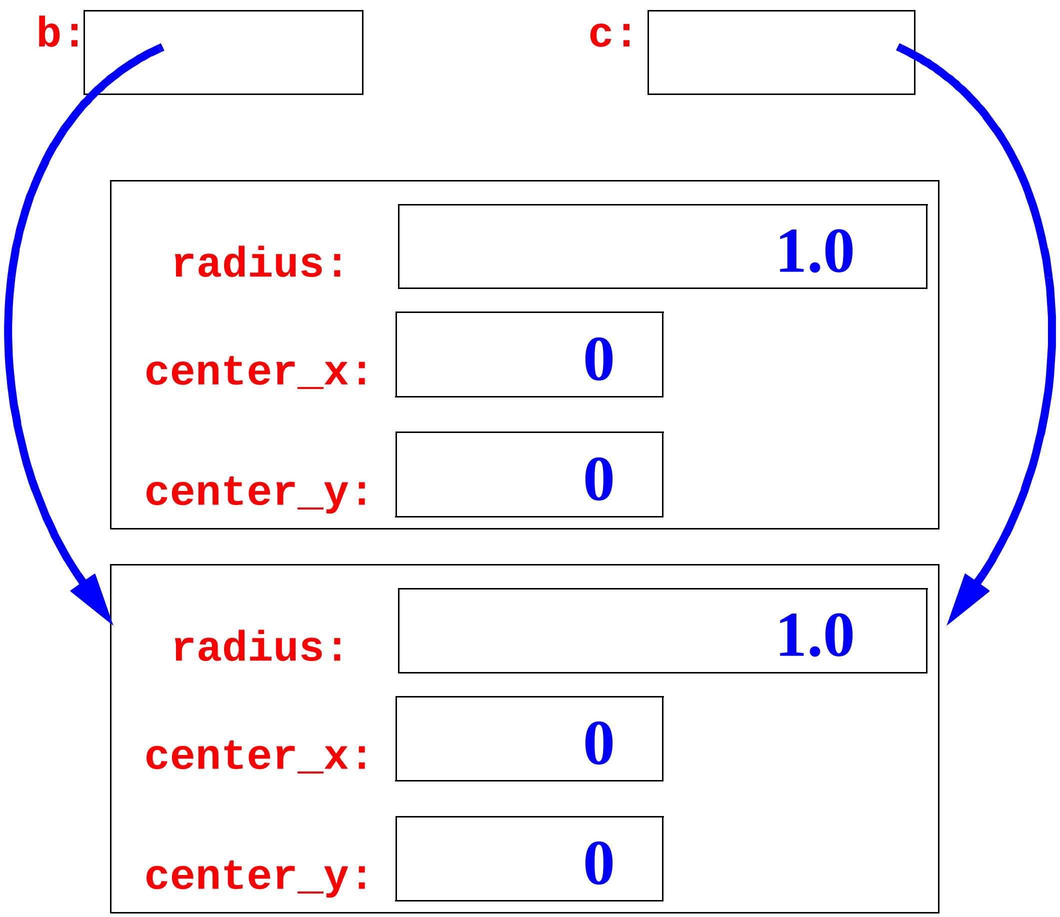
assign c to b.
Now b and c point to the same object

```
Circle b,c;

b = new Circle();

c = new Circle();

b = c;
```



Lifetime of objects: an example, frame 4

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

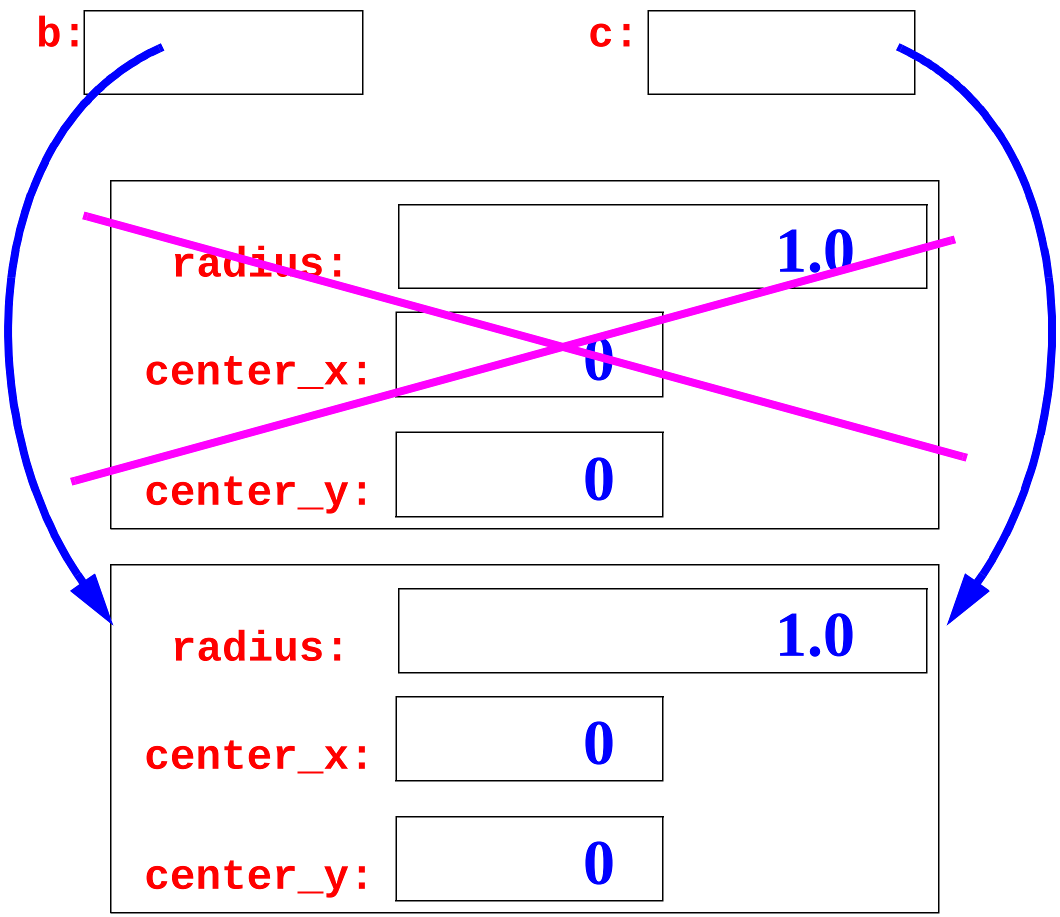
There is no pointer to the first object...
Its lifetime is over, and the GC can reclaim it

```
Circle b,c;

b = new Circle();

c = new Circle();

b = c;
```



Top-down software design

- A large problem can be broken down into a collection of subproblems
  - Each subproblem can be solved separately, and the solutions combined into an overall solution
  - (And the subproblems can be broken down into sub-subproblems, etc!)
  - Known as "top-down design", "stepwise refinement", "divide-and-conquer"...
- Top-down design is another example of abstraction in computer programming
  - By dividing the problem into subproblems that can be solved independently, you can concentrate on some details while abstracting away the rest
  - This is essential in making really large problems solvable
- Top-down design can apply in procedural programming, and OO programing

Top-down design in two paradigms

- In Procedural programming:
  - Each subproblem is solved by a designing a subprogram (also called procedure, subroutine, function)
  - These functions are called by the main() function and each other to solve the overall problem
- In OO programming:
  - Each subproblem is solved by designing a class
  - Objects that are instances of these classes are created and their instance methods are called by a main() method and each other, to solve the overall problem

Software testing

- ✔ Remember the 3 kinds of bugs: syntax errors, runtime errors, logic errors

- ✔ The compiler finds the first of these, but the other two require program testing to track down and fix

- ✔ Some rules of software testing:
  - ✗ Test every function in your program
  - ✗ Test every path of execution in every function (every branch of every if-else)
  - ✗ To attain "full-coverage" testing like this, you need to test on a variety of well-selected input data

Top-down and bottom-up testing with top-down design

- Recall the idea of "top-down", "stepwise refinement" design:
  - Start with a problem, decompose it into subproblems, decompose them into sub-subproblems, etc.
  - Each subproblem is solved by writing a class or a method that is called by methods at a "higher" level, and that calls methods at a "lower" level

- Every class and method should ultimately be tested in the context of the entire working system. But the pieces should be tested separately first. How to do that?

- "Bottom-up" testing of a method: The method will be called by other methods, which have not been written yet. So, write a *test driver* that will exercise the method, putting it through its paces in a simulation of how it might be used in the final working software system

- "Top-down" testing of a method: The method calls other methods, which have not been written yet. So, write *stubs* for these: methods which have the return type, name, and type of arguments of the real methods which will be called, but that don't do much (maybe just return a value or print a simple message) for testing purposes

Tracing and debugging

- When trying to track down logic or runtime errors in your program, it is helpful to look at the values of variables in your program as it is running
  - this is called *tracing* the variables

- A debugger is a program that lets you trace variables
  - jdb is the standard JDK Java debugger... but it is not really very easy to use
  - other development environments for Java may provide nice debuggers

- But instead of using a separate debugger program, you can always place statements in your program to print out the values of variables while testing and debugging it
  - remove these print statements after debugging!
  - hint: use the standard error output stream `System.err` instead of `System.out`
    - `err` also normally prints to the terminal, but `err` statements are distinctive, so easy to search for and remove when done with debugging

Tracing variables

- ✔ Useful places to put print statements for variable tracing:
  - ✗ as the first statement in a method body
    - lets you determine that the function is called
    - lets you print out the values of the formal arguments passed in to the function
  - ✗ just before a return statement in a method body
    - lets you print out the value returned from the function
  - ✗ as the first or last statement in a loop body
    - lets you see how many times the loop body is executed
    - lets you print out initial or final values of the variables in each loop execution
  - ✗ before and after an assignment statement
    - lets you see how the value of a variable changes
  - ✗ elsewhere as needed to indicate progress as your program executes

Example of tracing a function...

```
int computeStuff(int a1, double a2, char a3)
{
   System.err.print("Entering computeStuff: ");
   System.err.print(" a1=" + a1);
   System.err.print(" a2=" + a2);
   System.err.println(" a3=" + a3);

   int result;

   // ......



   System.err.println("Returning " + result +
                      " from computeStuff");

   return result;

}
```

Next time (after the exam)

- 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)