CSE 11: Lecture 5

- Variable visibility
- Arguments passed by value and by reference
- ADT's, visibility, and object-oriented design
- Equality testing with == and equals()
- Java wrapper classes
- Methods for parsing text

(Reading: Savitch, parts of Ch. 4 and Ch. 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

Variable visibility or scope

- A method must be defined before it can be used, and a variable must be declared before it can be used... but what does “before” mean?
- Like methods, variables have visibility rules, and a variable (like a method) can only be referred to from where its declaration is visible

- Visibility (sometimes called *scope*) rules for variables in Java:
  - a variable declared inside a compound statement (for example, a method body) is visible from the point of declaration to the end of the compound statement (it is “local” in that block)

  - a variable declared as a formal parameter in the argument list of a method header is visible everywhere in the method body (it is local in that method)

  - a variable declared static in a class outside any method body is visible everywhere in that class (it is “global in the class”), unless it is hidden within a method by a local variable declaration of the same name
    - if it is declared private, it is only visible in its class; otherwise it is visible outside the class according to visibility specifier rules
    - Outside the class, the variable name must be qualified with its class name

Formal arguments as variables

- Identifiers in the argument list of a method definition are the "formal arguments" or "formal parameters"

- These formal arguments act like variables that are visible only in the method body
  - so variables in the argument list can be used in the method body...
  - along with local variables declared in the method body...
  - along with variables declared static outside of any method body, in the same class as the method

Passing arguments in Java

- In Java, primitive-type arguments are always *passed by value*
  - The number and type of actual arguments in the call must match the number and type of formal arguments in the method header (subject to automatic type conversions which are basically the same as for assignment)
  - At the time of the method call, copies of the values of each actual argument is made available as the value of the corresponding formal argument
  - The formal argument is a different variable from anything in the actual argument list: it refers only to its copy of the value of the actual argument, so it cannot be used to change the value of the actual argument
- (In Java, objects are always *passed by reference* --- more about that soon)

Argument passed by value

```
public class Test {
static void foo(int a)  // when foo is called, value of actual
                        // argument is assigned to
                        // the formal argument a
{
   a = a + 1;                    // value of a is increased by 1
   System.out.println(a);     // prints out value of a
}


public static void main(String args[])
{
   int b = 3; // creates an int variable named b with value 3
   foo(b);    // calls function foo, passing the value of
              // the actual argument b

   System.out.println(b);     // prints out value of b, which is...

}
```

Argument passed by value: frame 1

```
public class Test {
static void foo(int a)


{
   a = a + 1;
   System.out.println(a);
}


public static void main(String args[])
{
   int b = 3;


   foo(b);

   System.out.println(b);

}
```

b: 3

Argument passed by value: frame 2

```
public class Test {
static void foo(int a)

                                                    a:  3


{
   a = a + 1;
   System.out.println(a);
}


public static void main(String args[])
{
   int b = 3;

                                                    b:  3


   foo(b);

   System.out.println(b);

}
```

Argument passed by value: frame 3

```
public class Test {
static void foo(int a)

{
   a = a + 1;
   System.out.println(a);
}


public static void main(String args[])
{
   int b = 3;

   foo(b);

   System.out.println(b);

}
```

a: 4

b: 3

Argument passed by value: frame 4

```
public class Test {
static void foo(int a)
```

a: 4

```
{
   a = a + 1;
   System.out.println(a);     // prints 4
}


public static void main(String args[])
{
   int b = 3;
```

b: 3

```
   foo(b);

   System.out.println(b);

}
```

Arguments passed by value: frame 5

```
public class Test {
static void foo(int a)



{
   a = a + 1;
   System.out.println(a);
}


public static void main(String args[])
{
   int b = 3;

                                                    b:  3

   foo(b);

   System.out.println(b);      // prints 3

}
```

Argument passed by value, another example

```
public class Test {
static void foo(int a)  // when foo is called, value of actual
                        // argument is copied as
                        // the value of the formal argument a
{
   a = a + 1;       // value of a is increased by 1
   System.out.println(a);       // prints out value of a
}


public static void main(String args[])
{
   int a = 3;// creates an int variable named b with value 3
   foo(a);// calls function foo, passing the value of
                    // the actual argument a

   System.out.println(a);     // prints out value of a, which is...

}
```

Review: Primitive and class types, assignment, and arguments

- Assignment has a somewhat different semantics for primitive types vs. objects:
  - if **a**, **b** are variables of primitive type, **a=b** copies **b**'s value into **a**'s storage location
  - if **a**,**b** are variables of reference type, **a=b** copies the address stored in **b** to **a**'s storage location, which makes **b** and **a** point to the same object

- Recall that in Java, when a method is called, values of actual arguments are in effect *assigned* to the corresponding formal parameters in the method definition

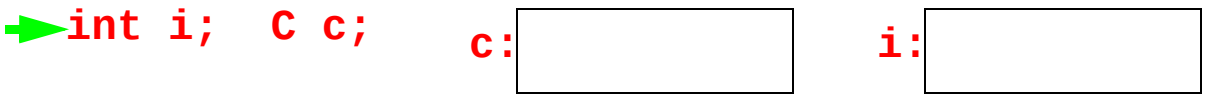
- This means that primitive types and objects behave differently when passed as arguments:
  - primitive types: the called method's formal parameter cannot be used to access the actual parameter. It only accesses a copy of the actual parameter's value
    - primitive types are passed by value
  - objects: the called method's formal parameter CAN be used to access the object the actual parameter points to... they are both pointers to the same object!
    - objects are passed by reference: the object pointed to is NOT copied

Argument passed by reference: an example

```
public class C {
   private int val;
   public void setVal(int v) {val = v;}
   public int getVal() {return val;}
}
-----------------------------------------------------
public class Test {
   private static void doIt (C c, int i) {
      i = i + 1;
      c.setVal(i);
   }

   public static void main (String[] args) {
      int i;  C c = new C();
      i = 4;
      c.setVal(4);
      doIt(c,i);

      System.out.println(i);                  // prints...
      System.out.println(c.getVal());         // prints...
```

Argument passed by reference: frame 1

```
public class Test {
 private static void doIt (C c, int i) {
 i = i + 1;
 c.setVal(i);
}

public static void main (String[] args) {
int i;  C c;
```

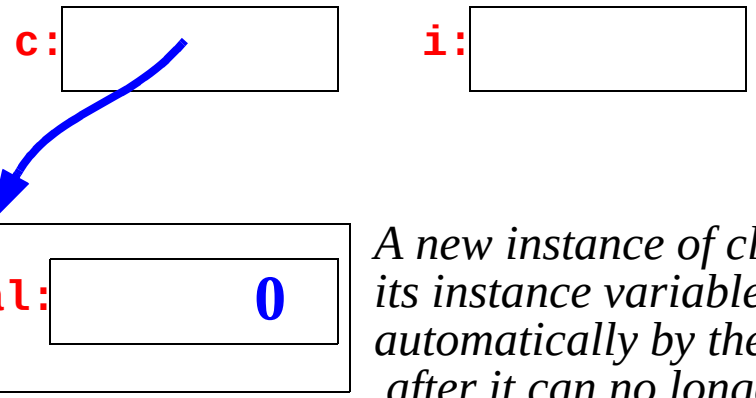


This `c` *and* `i` *are local in main(). They are created when their declarations are executed, and destroyed when main returns*

```
c = new C();

i = 4;
c.setVal(4);
doIt(c,i);

System.out.println(i);                  // prints...
System.out.println(c.getVal());         // prints...
```

Argument passed by reference: frame 2

```
public class Test {
 private static void doIt (C c, int i) {
 i = i + 1;
 c.setVal(i);
}

public static void main (String[] args) {
 int i;  C c;
```

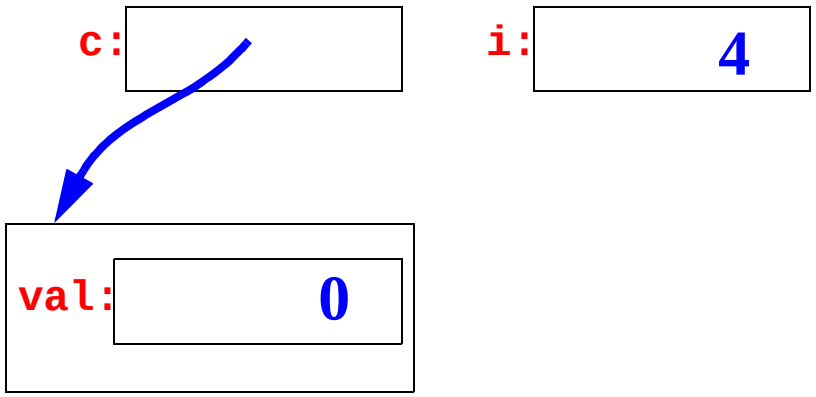



```
c = new C();

 i = 4;
 c.setVal(4);
 doIt(c,i);

 System.out.println(i);                  // prints...
 System.out.println(c.getVal());         // prints...
```

A new instance of class C is created, with its instance variables. It will be destroyed automatically by the garbage collector after it can no longer be accessed

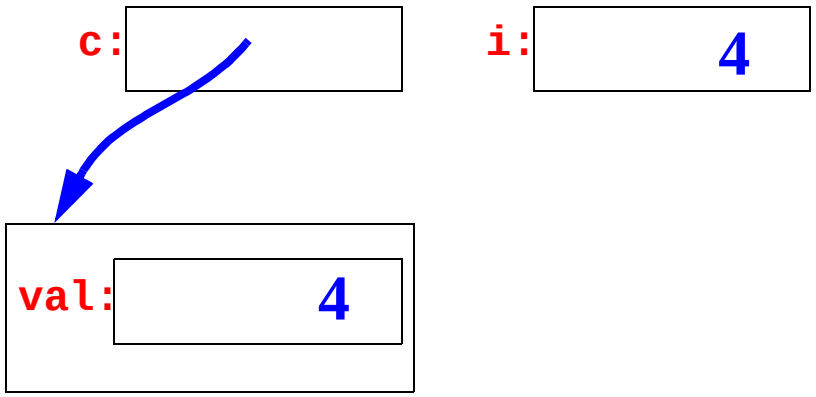
Argument passed by reference: frame 3

```
public class Test {
 private static void doIt (C c, int i) {
 i = i + 1;
 c.setVal(i);
}

public static void main (String[] args) {
 int i;  C c;

 c = new C();

 i = 4;
 c.setVal(4);
 doIt(c,i);

 System.out.println(i);                  // prints...
 System.out.println(c.getVal());         // prints...
```



Argument passed by reference: frame 4

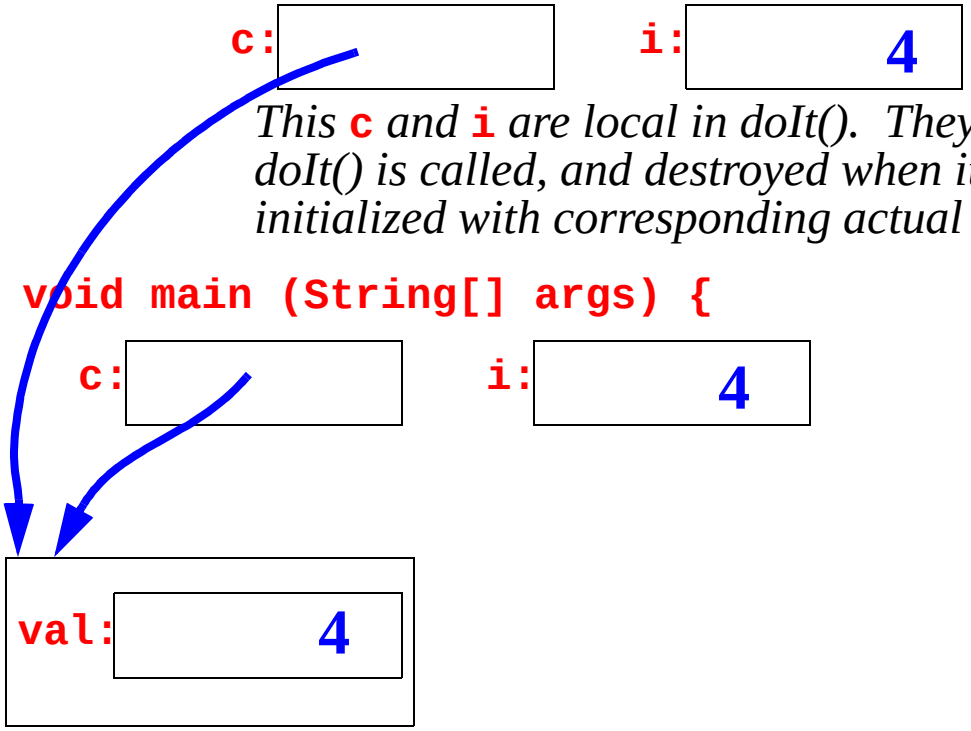
```
public class Test {
 private static void doIt (C c, int i) {
 i = i + 1;
 c.setVal(i);
}

public static void main (String[] args) {
 int i;  C c;

 c = new C();

 i = 4;
 c.setVal(4);
 doIt(c,i);

 System.out.println(i);                  // prints...
 System.out.println(c.getVal());         // prints...
```



Call setVal() instance method of object c

Argument passed by reference: frame 5

```
public class Test {
 private static void doIt (C c, int i) {
 i = i + 1;
 c.setVal(i);
}

public static void main (String[] args) {
 int i;  C c;

 c = new C();

 i = 4;
 c.setVal(4);
 doIt(c,i);

 System.out.println(i);                  // prints...
 System.out.println(c.getVal());         // prints...
```

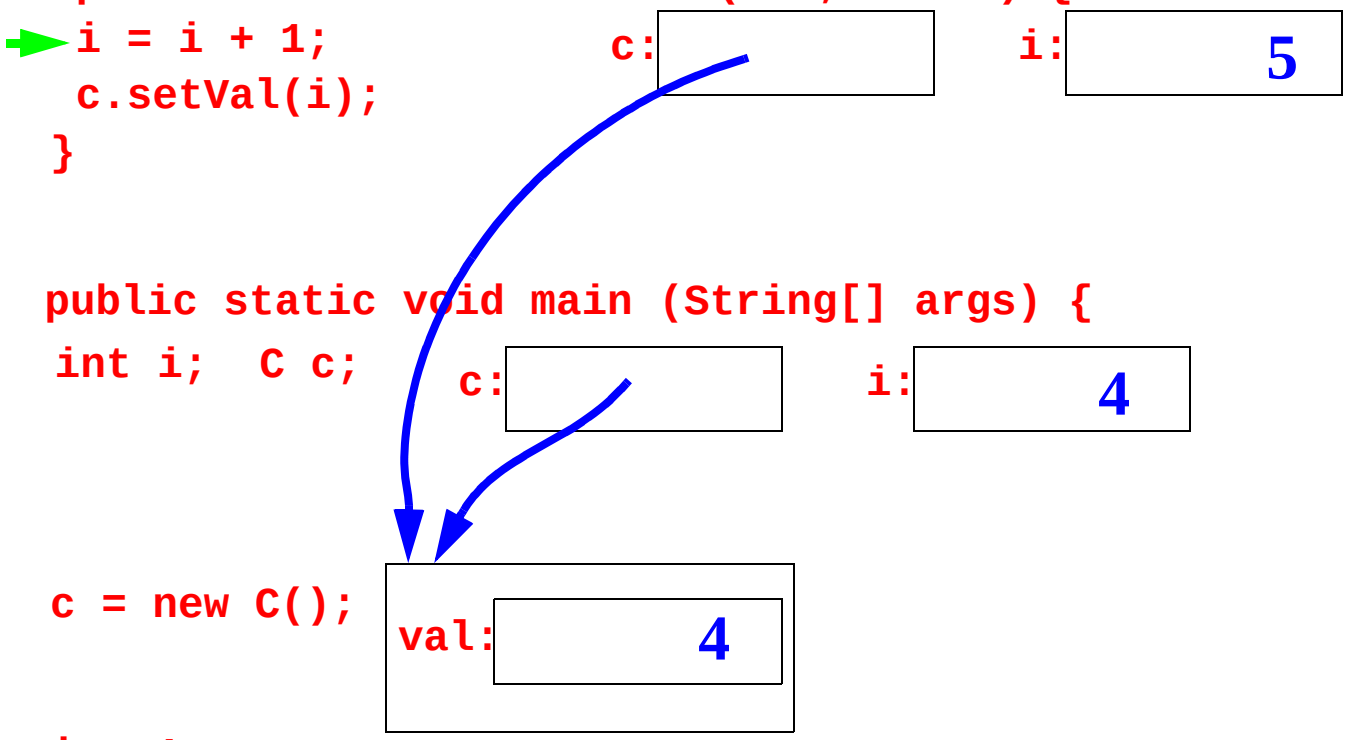


This `c` *and* `i` *are local in doIt(). They are created when doIt() is called, and destroyed when it returns. They are initialized with corresponding actual argument values*

Call doIt() static method, passing `c` *and* `i`

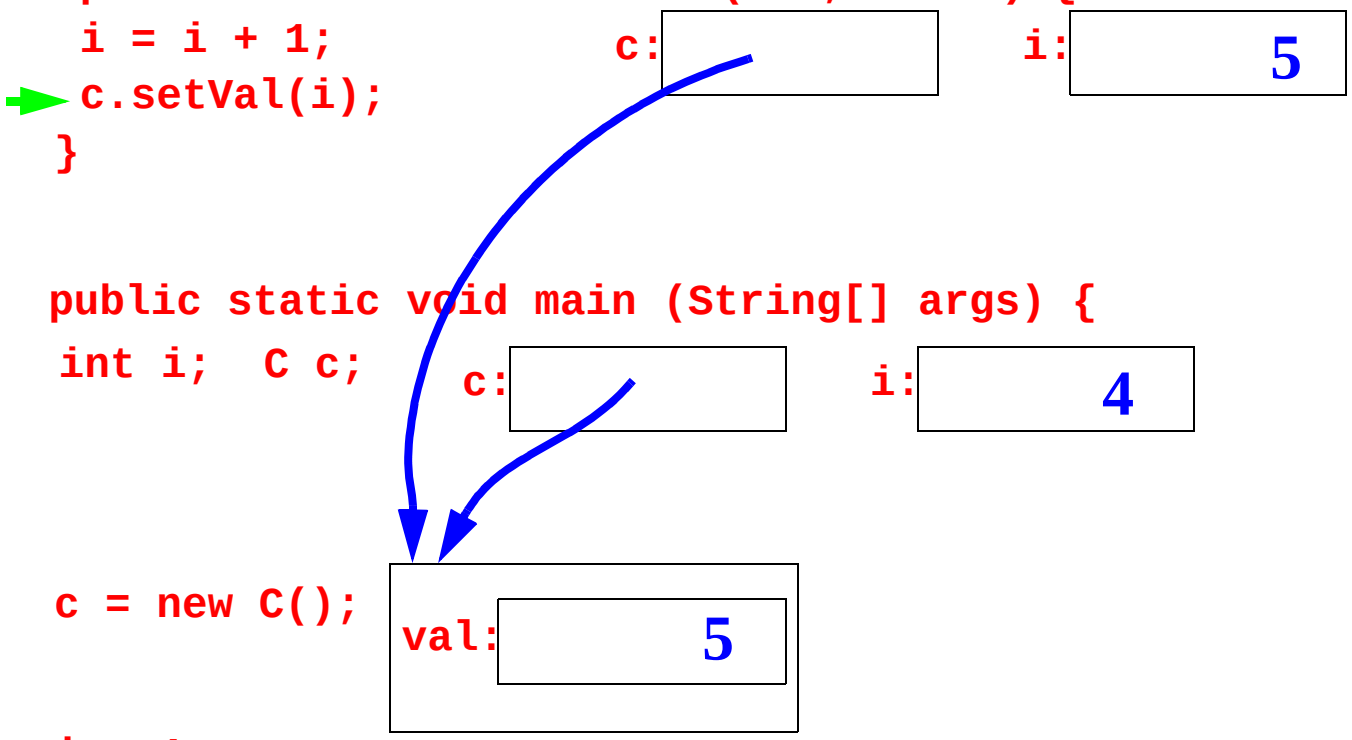
Argument passed by reference: frame 6

```
public class Test {
 private static void doIt (C c, int i) {
 i = i + 1;
 c.setVal(i);
}

public static void main (String[] args) {
 int i;  C c;

 c = new C();

 i = 4;
 c.setVal(4);
 doIt(c,i);

 System.out.println(i);                  // prints...
 System.out.println(c.getVal());         // prints...
```



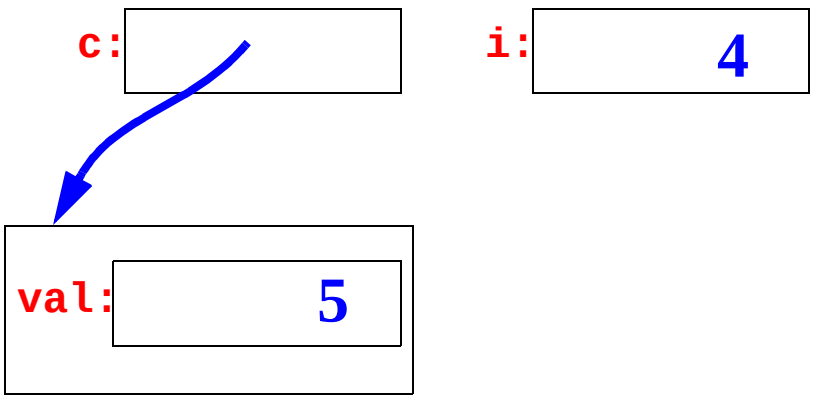
Argument passed by reference: frame 7

```
public class Test {
 private static void doIt (C c, int i) {
 i = i + 1;
 c.setVal(i);
}

public static void main (String[] args) {
 int i;  C c;

 c = new C();

 i = 4;
 c.setVal(4);
 doIt(c,i);

 System.out.println(i);                  // prints...
 System.out.println(c.getVal());         // prints...
```



Argument passed by reference: frame 8

```
public class Test {
 private static void doIt (C c, int i) {
 i = i + 1;
 c.setVal(i);
}


public static void main (String[] args) {
int i;  C c;
```




```
c = new C();


i = 4;
c.setVal(4);
doIt(c,i);
```

doIt() returns, and we continue...

```
System.out.println(i);                  // prints...
System.out.println(c.getVal());         // prints...
```

Argument passed by reference: another example

- Note this slightly different example:

```
public class Test {
   private static void doIt (C d, int j) {
      j = j + 1;
      d.setVal(j);
   }

   public static void main (String[] args) {
      int i;  C c;
      c = new C();
      i = 4;
      c.setVal(4);
      doIt(c,i);

      System.out.println(i);                  // prints...
      System.out.println(c.getVal());         // prints...
```

- Does this behave any differently from the previous example?

ADT's and classes

- Recall that a type is a collection of possible values, together with operations on those values; and an Abstract Data Type (ADT) is a data type that is designed using *abstraction*, i.e. hiding some details so the user can concentrate on more important ones
  - In an ADT, the implementations of the data values, and the operations on the data values, are designed as "black boxes"
  - So, the ADT can be successfully used without having to worry about the details of the implementation
- An object is something that encapsulates properties (i.e. data) and behavior (i.e. operations on that data), and an object gets its kind of data and behaviors from the definition of the class it is created as an instance of
- So in an OO language such as Java, it is natural to implement a user-defined ADT by defining a class!
- In designing an ADT, you have to make choices about what is part of the public interface to the ADT, and what will be hidden inside the "black box"
- These choices correspond to choice of visibility of members defined in the class that implements the ADT

State- and behavior-hiding in Object Oriented Design

- Some methods of an object should be accessible from "outside the object"
  - these are *public* methods: they are accessible with the dot operator whereever there is a pointer to the object
  - public member functions represent the possible behaviors of the object that can be invoked from outside the "black box"
  - they are part of the *user interface* to the object

- Normally, instance variables (and perhaps some methods) of an object should *not* be directly accessible from "outside the object"
  - these are *private* variables and methods:
    they are accessible only from instance methods of the object itself
  - private instance variables implement the state of the object (inside the "black box")
  - they are part of the *implementation* of the object
  - (use public instance methods to provide an interface to access the private variables if needed)

- Good ADT design, with limited access to object state and behavior, helps control bugs and makes both using and maintaining the code easier

More on ADT design in Java

- Rule of thumb: make all instance variables private. Our example **`Circle`** class does this

- But it is legal in Java, and for very simple classes it sometimes makes sense, to make instance variables public.
  - For example, the Point class in the Java AWT library does this:

```
/** A point representing a locaton in (x,y) coordinate space,
  * specified to int precision. */
public class Point {

   public int x;
   public int y;

   public Point(int x, int y) {
      this.x = x; this.y = y;
   }
}
```

Public and private visibility: examples

- When an object is created, it contains storage for all the instance variables specified within its class definition (the bytecode for instance methods is actually stored somewhere else, but conceptually they are attached to the object)

```
Circle c = new Circle();          // create a Circle object
Point mypt = new Point(3,4);      // create a Point object
```

- All public members of the object can be accessed using a pointer to the object, the dot operator '.', and the name of the member

```
c.setRadius(3.0);
System.out.println(c.AreaOf());
mynum.x = 100;
System.out.println(mynum.y);
```

- Private members of the object cannot be accessed using the dot operator, unless inside the body of a method of the object's class. Anywhere else, these will not compile:

```
c.radius = 3.0;    // illegal:  radius is private in c
int i = c.center_x;  // illegal:  center_x is private in c
```

Still more on ADT design with classes

- A well-designed class usually has these operations defined for objects of the class:
  - Initialization of data members (constructors)
  - Testing if two objects of the class are equal in value (define **`equals()`**)
  - Converting an object to a String, to represent the state of the object in printable form (define **`toString()`**)
  - Other basic operations suitable for the class, including accessor methods for inspecting and modifying the object's state (instance variables), as appropriate

Equality testing: == and the equals() method

- Recall that Java has these comparison operators that work on numeric type arguments:

  `<` `>` `<=` `>=` `==` `!=`

- The `==` and `!=` comparison operators will also work on values of boolean and reference type

- However, on reference type arguments, the `==` operator has value true (and the `!=` operator has value false) if and only if its arguments contain the same memory address, that is, they reference *exactly the same object*

- Often you are really more interested in whether two objects have the same properties, i.e. whether they *contain data that has the same value*

- To test for this, you should use one of the object's `equals()` instance method, and pass the other object as argument

- Every class in Java should define `equals()` to do equality testing as appropriate for objects of that type

Example: defining equals() for a Fraction class

```
public class Fraction {

   // initialize a Fraction object
   public Fraction(long num, long denom) {
      n = num; d = denom;
   }

   // return true if Fraction objects are equal
   public boolean equals(Fraction other) {
      // how to define this method?
   }

   private long n; // the numerator
   private long d; // the denominator
};
```

- How to define that **`equals`** method? First, think of an algorithm, based on the cross-product rule for fractions:

  $\frac{A}{B}$ equals $\frac{C}{D}$ if and only if AD equals BC

Testing Fractions for equality... with the equals() method

- Define the **`equals`** method, translating the idea of the algorithm into source code

```
// Equals returns true if argument Fraction
// is numerically equal to this Fraction
public boolean equals(Fraction other)
{
  return (this.n * other.d) == (this.d * other.n);
}
```

Using the equals() method

- With the **equals** instance method defined, it can be used to test for equality between instances of the class. For example:

```
public static void main(String args[])
{
    // create two Fraction objects and initialize them
    Fraction f1 = new Fraction(2,3);
    Fraction f2 = new Fraction(6,9);

    if( f1.equals(f2) )
        System.out.println("Yes they are equal.");
    else
        System.out.println("No they are not equal.");
}
```

A better version of the equals() method

- For reasons that we'll get into when we talk about class derivation, inheritance, and dynamic method binding, it is in fact better practice to define the the `equals` method to take one argument of type `Object`
- Since the argument is not declared to be of type `Fraction`, it needs to be cast before its `Fraction` instance members can be accessed
- The `instanceof` operator can be used to check if the argument is really pointing to a `Fraction` object
- Also we should check if the argument is null (if the method is called, the calling object cannot be null, but the argument can).
- The resulting method definition looks like:

```
public boolean equals(Object other) {
    // check for null argument
   if (other == null) return false;
   // check argument really points to a Fraction object
   if ( ! (other instanceof Fraction) ) return false;
   Fraction f = (Fraction) other;  // cast
   return (this.n * f.d) == (this.d * f.n);
}
```

Wrapper classes

- You know that Java makes a distinction between primitive types (byte, short, int, long, float, double, char, boolean) and reference types (everything else)
- For each of the primitive types, Java provides a corresponding class in the java.lang package:

```
Byte      <-->   byte
Short     <-->   short
Integer   <-->   int
Long      <-->   long
Float     <-->   float
Double    <-->   double
Character  <--> char
Boolean   <-->   boolean
```

- These are called “wrapper” classes, because an instance of one is designed to “wrap” (i.e., to contain as an instance variable) a value of its corresponding primitive type
- The wrapper classes give Java programmers a way to treat primitive type values as objects: by wrapping them in wrapper-class objects
- The wrapper classes also define static constants and methods that are useful for manipulating values of the primitive type

Using wrapper classes

- Each wrapper class has a constructor that takes one argument of its corresponding primitive type, and initializes the new instance to contain the value of that argument:

```
Integer myInt = new Integer(3);  // myInt wraps the int value 3
```

- Each wrapper class then has an instance method that returns the wrapped value. This method is named intValue for the Integer class, charValue for the Character class, etc:

```
int i = myInt.intValue();  // gets the wrapped value from myInt
```

- Each numerical wrapper class has constants (public static final variables) **MAX_VALUE** and **MIN_VALUE** which hold the largest and smallest possible values of the corresponding primitive type

```
System.out.println("The largest double is " + Double.MAX_VALUE);
```

- So far, this is not all that extremely useful. More useful methods coming up

Useful wrapper class methods

- Each wrapper class (except Character) has a public static method **`valueOf()`** which takes one argument of type String, and returns an instance of the class that wraps the primitive value represented by that String, interpreted as a literal constant of that primitive type. For example:

```
Double d = Double.valueOf("3.0");  // d wraps 3.0
System.out.println("d wraps " + d.doubleValue());

int k = 7 + Integer.valueOf("33").intValue();  // k ==  ??
```

- Each wrapper class also has a public instance method **`toString()`** which does essentially the inverse of the static method **`valueOf()`**

```
String s = d.toString();
System.out.println(s);            // prints... ??
```

- These methods are useful for parsing (breaking into parts and interpreting) arbitrary user input.

  (Savitch used them in writing SavitchIn.java...)

Static methods in the Character class

- The Character class has a number of static methods that take one argument of type char:

```
public static boolean isDigit (char ch)
public static boolean isLetter (char ch)
public static boolean isUpperCase (char ch)
public static boolean isLowerCase (char ch)
public static boolean isWhiteSpace (char ch)
public static char toLowerCase (char ch)
public static char toUpperCase (char ch)
```

- ... these can be useful for parsing text too

Parsing user input: an example

- The user will enter several integer literal constants on a line, separated by colons "**:**"
  Problem: Print the sum of the integers.
  Ex: user types **34 : -28 : 3 :111 <enter>**
  program prints: 120

```
String s = SavitchIn.readLine();  // read the whole line
int sum = 0;  // accumulate the sum here
int start =0, end=0;  // indexes into the String s

while(start < s.length()) {
   // find the next : or end of string
   while(end < s.length() && s.charAt(end) != ':') end++;
   // now start,end bracket a substring with no :'s
   // trim it, parse it as an int, and accumulate it in sum
   sum +=
      Integer.valueOf(s.substring(start,end).trim()).intValue();
   // set new start, end values
   start = end = end + 1;
}
System.out.println("The sum is " + sum);
```

Formatting output: an example

- Problem: Write a method **pad12** that takes a nonnegative int argument and returns a String of 12 digits representing the int, with as many leading zeros as necessary
  Ex: `System.out.println(pad12(777);` prints `000000000777`

```
String pad12(int num) {

   String numStr = (new Integer(num)).toString();

   int zerosNeeded = 12 - numStr.length();

   // build up a string of padding zeros
   String padStr = "";
   for(int i=0; i<zerosNeeded; i++) padStr += "0";

   return padStr + numStr;

}
```

Next time

- ✔ 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)