CSC 221: Computer Programming I

Fall 2006

Conditionals, expressions & modularization

- if statements, if-else
- increment/decrement, arithmetic assignments
- mixed expressions
- type casting
- abstraction, modularization
- internal vs. external method calls
- primitives vs. objects

Conditional execution

so far, all of the statements in methods have executed *unconditionally*

- when a method is called, the statements in the body are executed in sequence
- different parameter values may produce different results, but the steps are the same

many applications require *conditional execution*

- different parameter values may cause different statements to be executed

example: consider the `CashRegister` class

- previously, we assumed that method parameters were "reasonable"
  i.e., user wouldn't pay or purchase a negative amount
  user wouldn't check out unless payment amount ≥ purchase amount

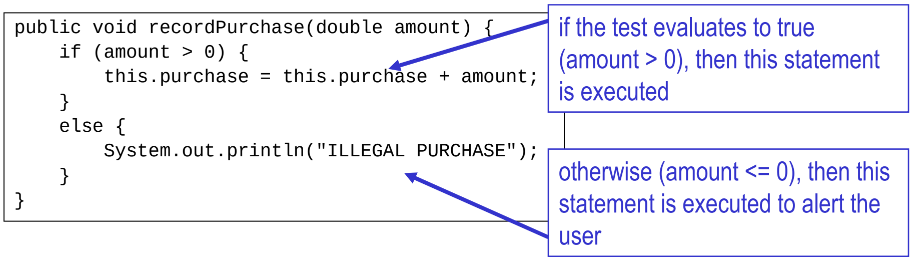
- to make this class more robust, we need to introduce conditional execution
  i.e., only add to purchase/payment total IF the amount is positive
  only allow checkout IF payment amount ≥ purchase amount

If statements

in Java, an *if statement* allows for conditional execution

- i.e., can choose between 2 alternatives to execute

```
if (perform some test) {
    Do the statements here if the test gave a true result
}
else {
    Do the statements here if the test gave a false result
}
```



```
public void recordPurchase(double amount) {
    if (amount > 0) {
        this.purchase = this.purchase + amount;
    }
    else {
        System.out.println("ILLEGAL PURCHASE");
    }
}
```

if the test evaluates to true (amount > 0), then this statement is executed

otherwise (amount <= 0), then this statement is executed to alert the user

If statements (cont.)

you are not required to have an else case to an if statement

- if no else case exists and the test evaluates to false, nothing is done
- e.g., could have just done the following

```
public void recordPurchase(double amount) {
    if (amount > 0) {
        this.purchase = this.purchase + amount;
    }
}
```

but then no warning to user if a negative amount were entered (not as nice)

standard relational operators are provided for the if test

<	less than	>	greater than
<=	less than or equal to	>=	greater than or equal to
==	equal to	!=	not equal to

a comparison using a relational operator is known as a *Boolean expression*, since it evaluates to a *Boolean* (`true` or `false`) value

In-class exercises

update `recordPurchase` to display an error message if attempt to purchase a negative or zero amount

```
public void recordPurchase(double amount) {
    if (amount > 0) {
        this.purchase = this.purchase + amount;
    }
    else {
        System.out.println("ILLEGAL PURCHASE");
    }
}
```

similarly, update `enterPayment`

In-class exercises

what changes should be made to `giveChange`?

```
public double giveChange() {
   if (_________________________) {
       double change = this.payment – this.purchase;

       this.purchase = 0;
       this.payment = 0;

       return change;
   }
   else {




   }
}
```

note: if a method has a non-void return type, every possible execution sequence must result in a return statement

- the Java compiler will complain otherwise

A further modification

suppose we wanted to add to the functionality of `CashRegister`

- get the number of items purchased so far
- get the average cost of purchased items

ADDITIONAL FIELDS?

CHANGES TO CONSTRUCTOR?

NEW METHODS?

Shorthand assignments

a variable that is used to keep track of how many times some event occurs is known as a *counter*

- a counter must be initialized to 0, then incremented each time the event occurs
- incrementing (or decrementing) a variable is such a common task that Java that Java provides a shorthand notation

`number++;`	is equivalent to	`number = number + 1;`
`number--;`	is equivalent to	`number = number - 1;`

other shorthand assignments can be used for updating variables

```
number += 5; ≡ number = number + 5;     number -= 1;  ≡ number = number - 1;

number *= 2; ≡ number = number * 2;     number /= 10; ≡ number = number / 10;
```

```
public void recordPurchase(double amount) {
    if (amount > 0) {
        this.purchase += amount;
    }
    else {
        System.out.println("ILLEGAL PURCHASE");
    }
}
```

Mixed expressions

note that when you had to calculate the average purchase amount, you divided the purchase total (`double`) with the number of purchases (`int`)

- mixed arithmetic expressions involving doubles and ints are acceptable
- in a mixed expression, the int value is automatically converted to a double and the result is a double

```
2 + 3.5   → 2.0 + 3.5 → 5.5

120.00 / 4 → 120.00 / 4.0 → 30.0

5 / 2.0 → 2.5
```

- however, if you apply an operator to two ints, you always get an int result

```
2 + 3   → 5

120 / 4 → 30

5 / 3 → 2 ???
```

CAREFUL: integer division throws away the fraction

Die revisited

extend the `Die` class to keep track of the *average* roll

- need a field to keep track of the total
- initialize the total in the constructors
- update the total on each roll
- compute the average by dividing the total with the number of rolls

```
public class Die {
    private int numSides;
    private int numRolls;
    private int rollTotal;

    public Die() {
      this.numSides = 6;
      this.numRolls = 0;
      this.rollTotal = 0;
    }

    public Die(int sides) {
      this.numSides = sides;
      this.numRolls = 0;
      this.rollTotal = 0;
    }

    public int getNumberOfSides() {
      return this.numSides;
    }

    public int getNumberOfRolls() {
      return this.numRolls;
    }

    public double getAverageOfRolls() {
      return this.rollTotal/this.numRolls;
    }

    public int roll() {
      this.numRolls++;

      int currentRoll = (int)(Math.random()*this.numSides + 1);
      this.rollTotal += currentRoll;

      return currentRoll;
    }
}
```

PROBLEM: since `rollTotal` and `numRolls` are both ints, integer division will be used

- avg of 1 & 2 will be 1

UGLY SOLUTION: make `rollTotal` be a double

- kludgy! it really is an int

Type casting

a better solution is to keep `rollTotal` as an int, but *cast* it to a double when needed

- casting tells the compiler to convert from one compatible type to another
- general form:
  `(NEW_TYPE)VALUE`
- if `rollTotal` is 3, the expression
  `(double)rollTotal`
  evaluates to 3.0

```
public class Die {
    private int numSides;
    private int numRolls;
    private int rollTotal;

    public Die() {
      this.numSides = 6;
      this.numRolls = 0;
      this.rollTotal = 0;
    }

    public Die(int sides) {
      this.numSides = sides;
      this.numRolls = 0;
      this.rollTotal = 0;
    }

    public int getNumberOfSides() {
      return this.numSides;
    }

    public int getNumberOfRolls() {
      return this.numRolls;
    }

    public double getAverageOfRolls() {
      return (double)this.rollTotal/this.numRolls;
    }

    public int roll() {
      this.numRolls++;

      int currentRoll = (int)(Math.random()*this.numSides + 1);
      this.rollTotal += currentRoll;

      return currentRoll;
    }
}
```

you can cast in the other direction as well (from a double to an int)

- any fractional part is lost
- if x is 3.7 → `(int)x` evaluates to 3

Complex expressions

how do you evaluate an expression like `1 + 2 * 3` and `8 / 4 / 2`

Java has rules that dictate the order in which evaluation takes place

- * and / have *higher precedence* than + and –, meaning that you evaluate the part involving * or / first

  `1 + 2 * 3` → `1 + (2 * 3)` → `1 + 6` → `7`

- given operators of the same precedence, you evaluate from left to right

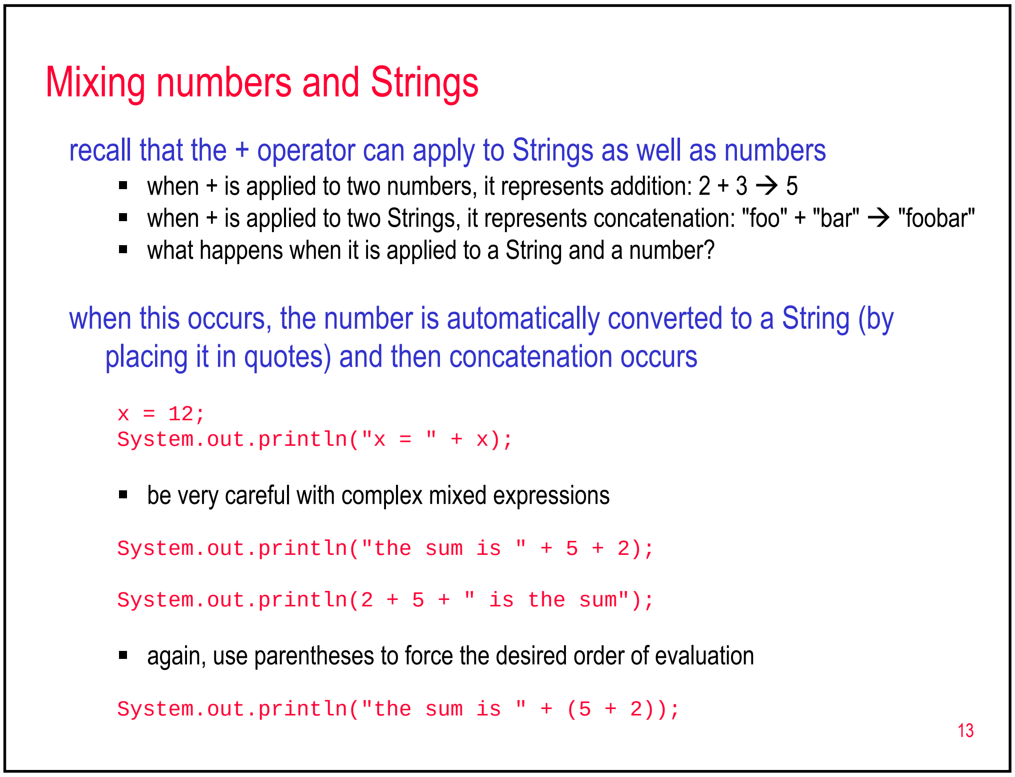
  `8 / 4 / 2` → `(8 / 4) / 2` → `2 / 2` → `1`

  `3 + 2 – 1` → `(3 + 2) – 1` → `5 – 1` → `4`

GOOD ADVICE: don't rely on these (sometimes tricky) rules

- place parentheses around sub-expressions to force the desired order

  `(3 + 2) – 1`   `3 + (2 – 1)`



Mixing numbers and Strings

recall that the + operator can apply to Strings as well as numbers

- when + is applied to two numbers, it represents addition: 2 + 3 → 5
- when + is applied to two Strings, it represents concatenation: "foo" + "bar" → "foobar"
- what happens when it is applied to a String and a number?

when this occurs, the number is automatically converted to a String (by placing it in quotes) and then concatenation occurs

```
x = 12;
System.out.println("x = " + x);
```

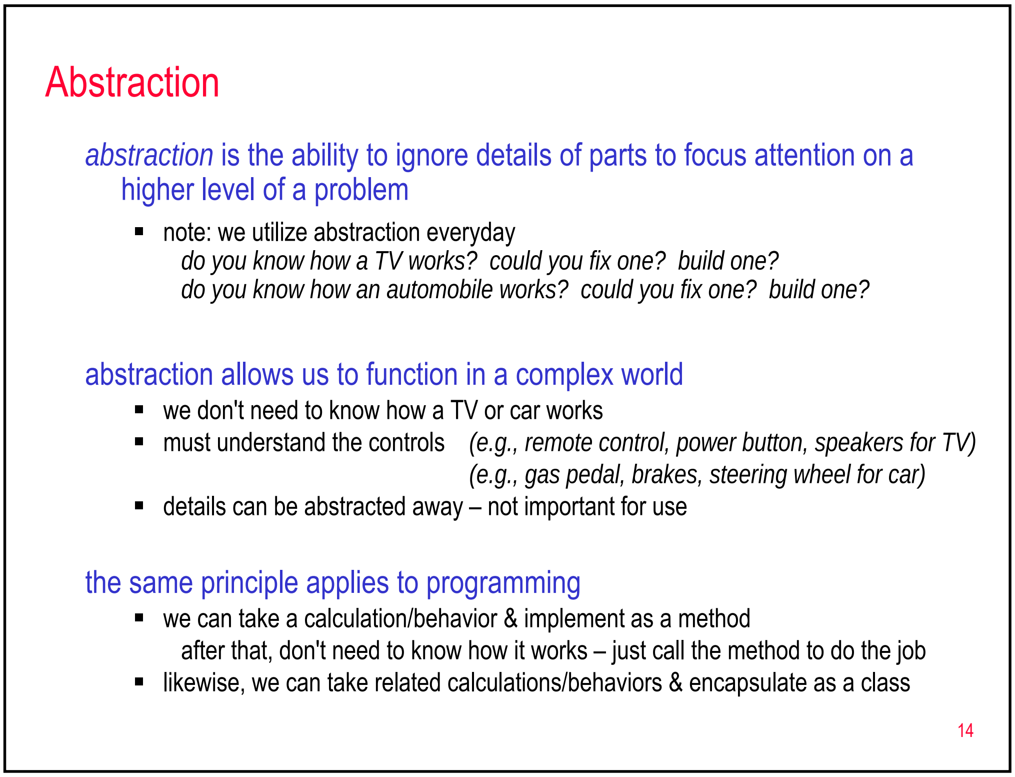
- be very careful with complex mixed expressions

```
System.out.println("the sum is " + 5 + 2);

System.out.println(2 + 5 + " is the sum");
```

- again, use parentheses to force the desired order of evaluation

```
System.out.println("the sum is " + (5 + 2));
```



Abstraction

abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem

- note: we utilize abstraction everyday
  do you know how a TV works? could you fix one? build one?
  do you know how an automobile works? could you fix one? build one?

abstraction allows us to function in a complex world

- we don't need to know how a TV or car works
- must understand the controls *(e.g., remote control, power button, speakers for TV)*
  (e.g., gas pedal, brakes, steering wheel for car)
- details can be abstracted away – not important for use

the same principle applies to programming

- we can take a calculation/behavior & implement as a method
  after that, don't need to know how it works – just call the method to do the job
- likewise, we can take related calculations/behaviors & encapsulate as a class

Abstraction examples

recall the Die class

- included the method `roll`, which returned a random roll of the Die

do you remember the formula for selecting a random number from the right range?

WHO CARES?!? Somebody figured it out once, why worry about it again?

SequenceGenerator class

- included the method `randomSequence`, which returned a random string of letters

you don't know enough to code it, but you could use it!

Circle, Square, Triangle classes

- included methods for drawing, moving, and resizing shapes

again, you don't know enough to code them, but you could use them!

Modularization

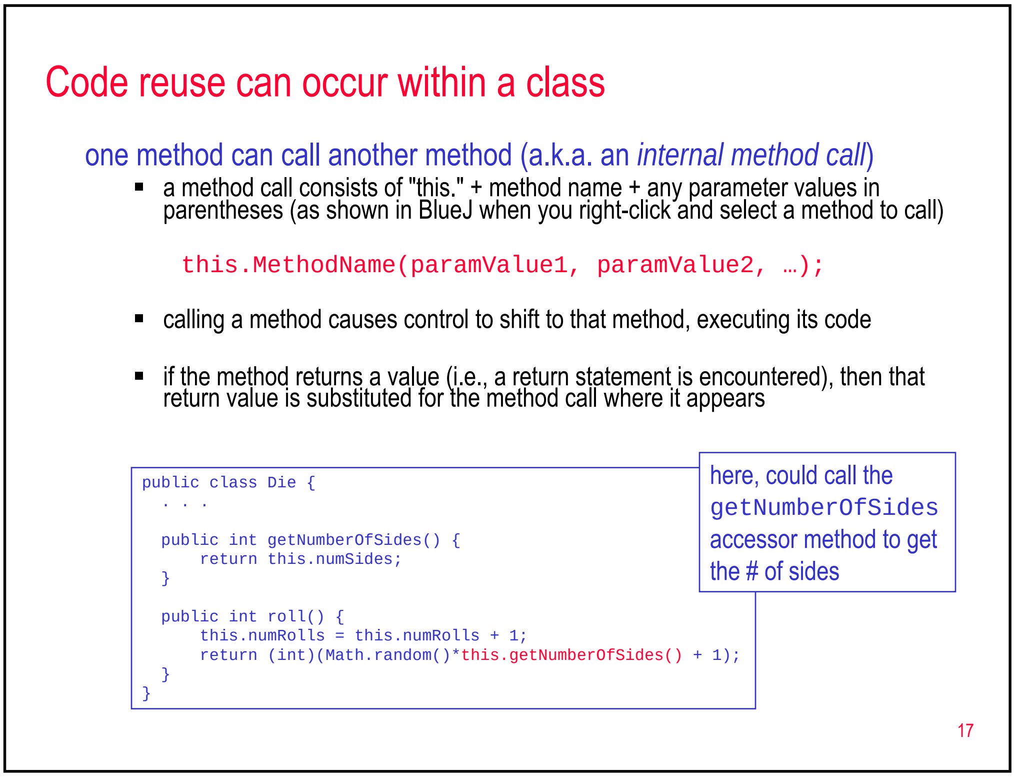
modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways

- early computers were hard to build – started with lots of simple components (e.g., vacuum tubes or transistors) and wired them together to perform complex tasks
- today, building a computer is relatively easy – start with high-level modules (e.g., CPU chip, RAM chips, hard drive) and plug them together

think Garanimals!

the same advantages apply to programs

- if you design and implement a method to perform a well-defined task, can call it over and over within the class
- likewise, if you design and implement a class to model a real-world object's behavior, then you can reuse it whenever that behavior is needed (e.g., Die for random values)



Code reuse can occur within a class

one method can call another method (a.k.a. an *internal method call*)

- a method call consists of "this." + method name + any parameter values in parentheses (as shown in BlueJ when you right-click and select a method to call)

```
this.MethodName(paramValue1, paramValue2, …);
```

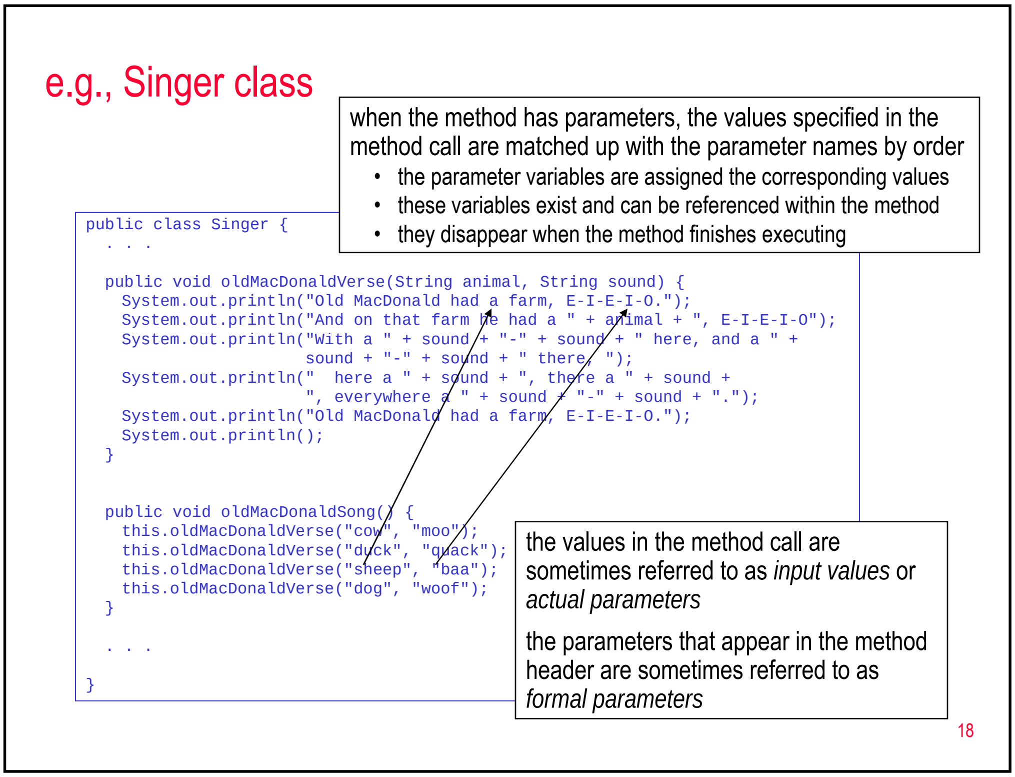
- calling a method causes control to shift to that method, executing its code
- if the method returns a value (i.e., a return statement is encountered), then that return value is substituted for the method call where it appears

```
public class Die {
  . . .

  public int getNumberOfSides() {
      return this.numSides;
  }

  public int roll() {
      this.numRolls = this.numRolls + 1;
      return (int)(Math.random()*this.getNumberOfSides() + 1);
  }
}
```

here, could call the `getNumberOfSides` accessor method to get the # of sides



e.g., Singer class

when the method has parameters, the values specified in the method call are matched up with the parameter names by order

- the parameter variables are assigned the corresponding values
- these variables exist and can be referenced within the method
- they disappear when the method finishes executing

```
public class Singer {
  . . .

  public void oldMacDonaldVerse(String animal, String sound) {
    System.out.println("Old MacDonald had a farm, E-I-E-I-O.");
    System.out.println("And on that farm he had a " + animal + ", E-I-E-I-O");
    System.out.println("With a " + sound + "-" + sound + " here, and a " +
                       sound + "-" + sound + " there, ");
    System.out.println("  here a " + sound + ", there a " + sound +
                       ", everywhere a " + sound + "-" + sound + ".");
    System.out.println("Old MacDonald had a farm, E-I-E-I-O.");
    System.out.println();
  }


  public void oldMacDonaldSong() {
    this.oldMacDonaldVerse("cow", "moo");
    this.oldMacDonaldVerse("duck", "quack");
    this.oldMacDonaldVerse("sheep", "baa");
    this.oldMacDonaldVerse("dog", "woof");
  }

  . . .
}
```

the values in the method call are sometimes referred to as *input values* or *actual parameters*

the parameters that appear in the method header are sometimes referred to as *formal parameters*

Primitive types vs. object types

primitive types are predefined in Java, e.g., `int`, `double`, `boolean`, `char`

object types are those defined by classes, e.g., `Circle`, `Die`, `Singer`

- so far, our classes have utilized primitives for fields/parameters/local variables
- as we define classes that encapsulate useful behaviors, we will want to use them in other classes (e.g., have a Die field within a craps game class)

when you declare a variable of primitive type, memory is allocated for it

- to store a value, simply assign that value to the variable

```
int x;                          double height = 72.5;
x = 0;
```

when you declare a variable of object type, it is NOT automatically created

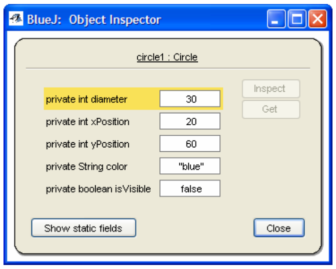
- to initialize, must call its constructor: `OBJECT = new CLASS(`*`PARAMETERS`*`);`
- to call a method: `OBJECT.METHOD(`*`PARAMETERS`*`)` (a.k.a. *external method call*)

```
Circle circle1;                 Die d8 = new Die(8);
circle1 = new Circle();         System.out.println( d8.roll() );
circle1.changeColor("red");
```

primitive types vs. object types

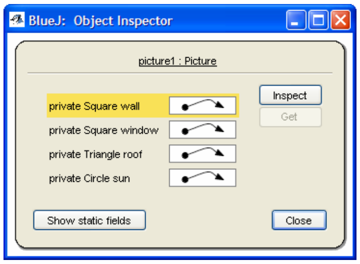
internally, primitive and reference types are stored differently

- when you inspect an object, any primitive fields are shown as boxes with values
- when you inspect an object, any object fields are shown as pointers to other objects



circle1 : Circle

private int diameter	30
private int xPosition	20
private int yPosition	60
private String color	"blue"
private boolean isVisible	false



picture1 : Picture

private Square wall

private Square window

private Triangle roof

private Circle sun

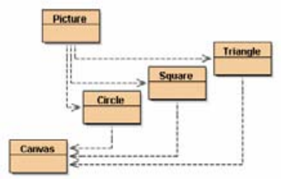
- of course, you can further inspect the contents of object fields

we will consider the implications of primitives vs. objects later

Picture example

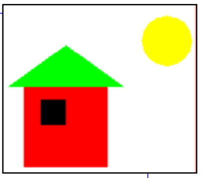
recall the Picture class, whose `draw` method automated the process of drawing the picture

- the class has fields for each of the shapes in the picture *(see class diagram for dependencies)*



- in the `draw` method, each shape is created by calling its constructor and assigning to the field

- then, methods are called on the shape objects to draw the scene



```
public class Picture {
    private Square wall;
    private Square window;
    private Triangle roof;
    private Circle sun;

    . . .

    public void draw() {
        this.wall = new Square();
        this.wall.moveVertical(80);
        this.wall.changeSize(100);
        this.wall.makeVisible();

        this.window = new Square();
        this.window.changeColor("black");
        this.window.moveHorizontal(20);
        this.window.moveVertical(100);
        this.window.makeVisible();

        this.roof = new Triangle();
        this.roof.changeSize(50, 140);
        this.roof.moveHorizontal(60);
        this.roof.moveVertical(70);
        this.roof.makeVisible();

        this.sun = new Circle();
        this.sun.changeColor("yellow");
        this.sun.moveHorizontal(180);
        this.sun.moveVertical(-10);
        this.sun.changeSize(60);
        this.sun.makeVisible();
    }
}
```

TEST 1

will contain a mixture of question types, to assess different kinds of knowledge

- quick-and-dirty, factual knowledge
  e.g., TRUE/FALSE, multiple choice — *similar to questions on quizzes*
- conceptual understanding
  e.g., short answer, explain code — *similar to quizzes, possibly deeper*
- practical knowledge & programming skills
  trace/analyze/modify/augment code — *either similar to homework exercises or somewhat simpler*

the test will contain several "extra" points

e.g., 52 or 53 points available, but graded on a scale of 50 (hey, mistakes happen ☺)

study advice:

- see online review sheet for outline of topics covered
- review lecture notes (if not *mentioned* in notes, will not be on test)
- read text to augment conceptual understanding, see more examples & exercises
- review quizzes and homeworks

- feel free to review other sources (lots of Java tutorials online)