

# CSC 221: Computer Programming I

Fall 2005

## interacting objects

- abstraction, modularization
- internal vs. external method calls
- primitives vs. objects
- modular design: dot races
- constants, static fields
- cascading if-else, logical operators
- variable scope

1

## 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

2

## 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!*

3

## 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)

4

## 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 method name + any parameter values in parentheses (as shown in BlueJ when you right-click and select a method to call)

```
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 numSides;
    }

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

here, the number returned by the call to `getNumberOfSides` is used to generate the random roll

5

## e.g., Singer class

```
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()
    {
        oldMacDonaldVerse("cow", "moo");
        oldMacDonaldVerse("duck", "quack");
        oldMacDonaldVerse("sheep", "baa");
        oldMacDonaldVerse("dog", "woof");
    }

    . . .
}
```

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

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*

6

## 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*)

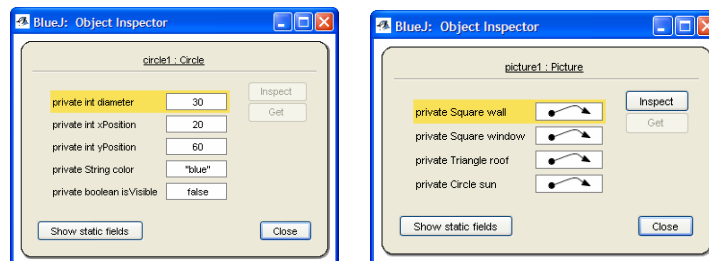
```
Circle circle1;                       Die d8 = new Die(8);
circle1 = new Circle();                System.out.println( d8.roll() );
circle1.changeColor("red");
```

7

## 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



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

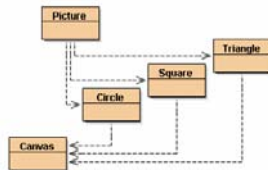
we will consider the implications of primitives vs. objects later

8

## 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;

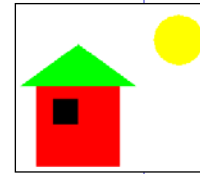
    . . .

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

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

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

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



9

## Dot races

consider the task of simulating a dot race (as on stadium scoreboards)

- different colored dots race to a finish line
- at every step, each dot moves a random distance (say between 1 and 5 units)
- the dot that reaches the finish line first wins!

behaviors?

- create a race (dots start at the beginning)
- step each dot forward a random amount
- access the positions of each dot
- display the status of the race
- reset the race

we could try modeling a race by implementing a class directly

- store positions of the dots in fields
- have each method access/update the dot positions

BUT: lots of details to keep track of; not easy to generalize

10

## A modular design

instead, we can encapsulate all of the behavior of a dot in a class

**Dot class:** create a `Dot` (with a given color)  
access the dot's position  
take a step  
reset the dot back to the beginning  
display the dot's color & position

once the `Dot` class is defined, a `DotRace` will be much simpler

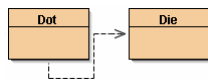
**DotRace class:** create a `DotRace` (with two dots)  
access either dot's position  
move both dots a single step  
reset both dots back to the beginning  
display both dots' color & position

11

## Dot class

more naturally:

- fields store a `Die` (for generating random steps), color & position



- constructor creates the `Die` object and initializes the color and position fields
- methods access and update these fields to maintain the dot's state

CREATE AND PLAY

```
public class Dot
{
    private Die die;
    private String dotColor;
    private int dotPosition;

    public Dot(String color)
    {
        die = new Die(5);
        dotColor = color;
        dotPosition = 0;
    }

    public int getPosition()
    {
        return dotPosition;
    }

    public void step()
    {
        dotPosition += die.roll();
    }

    public void reset()
    {
        dotPosition = 0;
    }

    public void showPosition()
    {
        System.out.println(dotColor +
            ": " + dotPosition);
    }
}
```

12

## Magic numbers

the Dot class works OK, but what if we wanted to change the range of a dot?

- e.g., instead of a step of 1-5 units, have a step of 1-8
- would have to go and change  
`die = new Die(5);`  
to  
`die = new Die(8);`
- having "magic numbers" like 5 in code is bad practice
  - ✓ unclear what 5 refers to when reading the code
  - ✓ requires searching for the number when a change is desired

```
public class Dot
{
    private Die die;
    private String dotColor;
    private int dotPosition;

    public Dot(String color)
    {
        die = new Die(5);
        dotColor = color;
        dotPosition = 0;
    }

    public int getPosition()
    {
        return dotPosition;
    }

    public void step()
    {
        dotPosition += die.roll();
    }

    public void reset()
    {
        dotPosition = 0;
    }

    public void showPosition()
    {
        System.out.println(dotColor +
            ": " + dotPosition);
    }
}
```

13

## Constants

better solution: define a *constant*

- a constant is a variable whose value cannot change
- use a constant any time a "magic number" appears in code

a constant declaration looks like any other variable except

- the keyword `final` specifies that the variable, once assigned a value, is unchangeable
- the keyword `static` specifies that the variable is shared by all objects of that class
  - since a final value cannot be changed, it is wasteful to have every object store a copy of it
  - instead, can have one static variable that is shared by all

*by convention, constants are written in all upper-case with underscores*

```
public class Dot
{
    private static final int MAX_STEP = 5;

    private Die die;
    private String dotColor;
    private int dotPosition;

    public Dot(String color)
    {
        die = new Die(MAX_STEP);
        dotColor = color;
        dotPosition = 0;
    }

    public int getPosition()
    {
        return dotPosition;
    }

    public void step()
    {
        dotPosition += die.roll();
    }

    public void reset()
    {
        dotPosition = 0;
    }

    public void showPosition()
    {
        System.out.println(dotColor +
            ": " + dotPosition);
    }
}
```

14

## Static fields

in fact, it is sometimes useful to have static fields that aren't constants

- there is no reason for every dot to have its own Die
- we could declare the Die field to be static, so that the one Die is shared by all dots

note: methods can be declared static as well

- e.g., `random` is a static method of the predefined `Math` class
- you call a static method by specifying the class name as opposed to an object name: `Math.random()`
- MORE LATER

```
public class Dot
{
    private static final int MAX_STEP = 5;

    private static Die die;
    private String dotColor;
    private int dotPosition;

    public Dot(String color)
    {
        die = new Die(MAX_STEP);
        dotColor = color;
        dotPosition = 0;
    }

    public int getPosition()
    {
        return dotPosition;
    }

    public void step()
    {
        dotPosition += die.roll();
    }

    public void reset()
    {
        dotPosition = 0;
    }

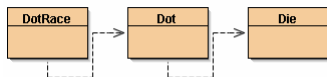
    public void showPosition()
    {
        System.out.println(dotColor +
            ": " + dotPosition);
    }
}
```

15

## DotRace class

using the `Dot` class, a `DotRace` class is straightforward

- fields store the two Dots



- constructor creates the `Dot` objects, initializing their colors and max steps
- methods utilize the `Dot` methods to produce the race behaviors

CREATE AND PLAY

ADD ANOTHER DOT?

```
public class DotRace
{
    private Dot redDot;
    private Dot blueDot;

    public DotRace()
    {
        redDot = new Dot("red");
        blueDot = new Dot("blue");
    }

    public int getRedPosition()
    {
        return redDot.getPosition();
    }

    public int getBluePosition()
    {
        return blueDot.getPosition();
    }

    public void step()
    {
        redDot.step();
        blueDot.step();
    }

    public void showStatus()
    {
        redDot.showPosition();
        blueDot.showPosition();
    }

    public void reset()
    {
        redDot.reset();
        blueDot.reset();
    }
}
```

16

## Adding a finish line

suppose we wanted to place a finish line on the race

- what changes would we need?

could add a field to store the goal distance

- user specifies goal distance along with max step size in constructor call
- `step` method would not move if either dot has crossed the finish line

```
public class DotRace
{
    private Dot redDot;
    private Dot blueDot;
    private int goalDistance; // distance to the finish line

    public DotRace(int goal)
    {
        redDot = new Dot("red");
        blueDot = new Dot("blue");
        goalDistance = goal;
    }

    public int getGoalDistance()
    {
        return goalDistance;
    }

    . . .
}
```

17

## Checking the finish line

to run a step of the race, really need to consider 3 possibilities

- RED has crossed the finish line
- BLUE has crossed the finish line
- neither has crossed, so the race must continue

recall: a simple if statement is a 1-way conditional (*execute this or not*)

```
if (redDot.getPosition() >= goalDistance) {
    System.out.println("The race is over!");
}
```

an if statement with else case is a 2-way conditional (*execute this or that*)

```
if (redDot.getPosition() >= goalDistance) {
    System.out.println("The race is over!");
}
else {
    // HANDLE THE CASE WHERE RED HASN'T CROSSED
}
```

here, the case where RED hasn't crossed must be broken into cases (BLUE has crossed, or not)

18

## Multi-way conditionals

can get a multi-way conditional by nesting if statements

```
public void step()
{
    if (redDot.getPosition() >= goalDistance) {
        System.out.println("The race is over!");
    }
    else {
        if (blueDot.getPosition() >= goalDistance) {
            System.out.println("The race is over!");
        }
        else {
            redDot.step();
            blueDot.step();
        }
    }
}
```

in fact, you can handle an arbitrary number of cases by nesting ifs

- deep nesting can lead to ugly code, however

19

## Cascading if-else

to make nested if statements look nicer, some curly-braces can be omitted and the cases indented to show structure

```
public void step()
{
    if (redDot.getPosition() >= goalDistance) {
        System.out.println("The race is over!");
    }
    else if (blueDot.getPosition() >= goalDistance) {
        System.out.println("The race is over!");
    }
    else {
        redDot.step();
        blueDot.step();
    }
}
```

such a structure is known as a *cascading if-else*

- control cascades down the cases like water cascading down a waterfall
- if a test fails, control moves down to the next one

```
if (TEST_1) {
    STATEMENTS_1;
}
else if (TEST_2) {
    STATEMENTS_2;
}
else if (TEST_3) {
    STATEMENTS_3;
}
. . .
else {
    STATEMENTS_ELSE;
}
```

20

## Combining cases

this code works, but contains redundancy

- there are two cases that result in the same code!

```
public void step()
{
    if (blueDot.getPosition() >= goalDistance) {
        System.out.println("The race is over!");
    }
    else if (redDot.getPosition() >= goalDistance) {
        System.out.println("The race is over!");
    }
    else {
        redDot.step();
        blueDot.step();
    }
}
```

fortunately, Java provides *logical operators* for simplifying such cases

(TEST1 || TEST2) evaluates to true if either TEST1 **OR** TEST2 is true

(TEST1 && TEST2) evaluates to true if either TEST1 **AND** TEST2 is true

(!TEST) evaluates to true if TEST is **NOT** true

21

## Logical operators

here, could use || to avoid duplication

- print message if *either* blue *or* red has crossed the finish line

```
public void step()
{
    if (redDot.getPosition() >= goalDistance || blueDot.getPosition() >= goalDistance) {
        System.out.println("The race is over!");
    }
    else {
        redDot.step();
        blueDot.step();
    }
}
```

**warning:** the tests that appear on both sides of || and && must be complete Boolean expressions

(x == 2 || x == 12) OK

(x == 2 || 12) BAD!

note: we could have easily written step using &&

- move dots if *both* blue *and* red dots have failed to cross finish line

```
public void step()
{
    if (redDot.getPosition() < goalDistance && blueDot.getPosition() < goalDistance) {
        redDot.step();
        blueDot.step();
    }
    else {
        System.out.println("The race is over!");
    }
}
```

22

## EXERCISE

how would we change `step` if we wanted to display the winner?

- RED wins!
- BLUE wins!
- It's a tie!

```
public void step()
{
    if ( ??? ) {
        .
        .
    }
    else {
        redDot.step();
        blueDot.step();
    }
}
```

23

## Modularity and scope

key idea: independent modules can be developed independently

- in the real world, a software project might get divided into several parts
- each part is designed & coded by a different team, then integrated together
- internal naming conflicts should not be a problem  
e.g., when declaring a local variable in a method, the programmer should not have to worry about whether that name is used elsewhere

in Java, all variables have a *scope* (i.e., a section of the program where they exist and can be accessed)

- the scope of a field is the entire class (i.e., all methods can access it)
- the scope of a parameter is its entire method
- the scope of a local variable is from its declaration to the end of its method
  
- so, you can use the same name as a local variable/parameter in multiple methods
- you can also use the same field name in different classes
- in fact, different classes may have methods with the same name!

24

## Ugly scope example

this class is legal (although bad style)

- try to avoid having parameters/locals with the same name as a field
- the compiler has no trouble distinguishing, but a human reader might

```
public class Ugly
{
    private int num;

    public Ugly()
    {
        num = 5;
    }

    public int method1(int num)
    {
        return num * 2           // refers to the parameter
    }

    public void method2()
    {
        int num = 0;
        System.out.println(num); // refers to the local
    }

    public void method3()
    {
        return num;             // refers to the field
    }
}
```

25

## If statements and scope

the curly braces associated with if statements also define new scopes

- a variable declared inside an if case is local to that case
- memory is allocated for the variable only if it is reached
- when that case is completed, the associated memory is reclaimed

```
public double giveChange()
{
    if (payment >= purchase {
        double change = payment - purchase;

        purchase = 0;
        payment = 0;

        return change;
    }
    else {
        System.out.println("Enter more money first.");
        return 0.0;
    }
}
```

if the test fails, then the declaration is never reached and the effort of creating & reclaiming memory is saved

also, cleaner since the variable is declared close to where it is needed

26

## Thursday: 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:

- 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)