CSC 221: Computer Programming I

Fall 2004

interacting objects

- abstraction, modularization
- internal method calls
- object creation, external method calls
- primitives vs. objects
- class diagrams
- modular design: dot races
- static fields

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 it, but you could use it!

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 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

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()
  {
    oldMacDonaldVerse("cow", "moo");
    oldMacDonaldVerse("duck", "quack");
    oldMacDonaldVerse("sheep", "baa");
    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*

SequenceGenerator class

```
public class SequenceGenerator
{
  . . .

  public String randomSequence(int len)
  {
    // code for generating and returning a random sequence of len letters
  }


  public void displaySequences(int seqLength)
  {
     System.out.println(randomSequence(seqLength) + " " + randomSequence(seqLength) + " " +
                        randomSequence(seqLength) + " " + randomSequence(seqLength) + " " +
                        randomSequence(seqLength));
   }

}
```

here, the code for generating a random sequence is abstracted away in the `randomSequence` method

can call that method multiple times in `displaySequences`, concatenate and display the strings

primitive types vs. object types

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

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 build on them

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;
x = 0;                       height = 72.5;
```

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

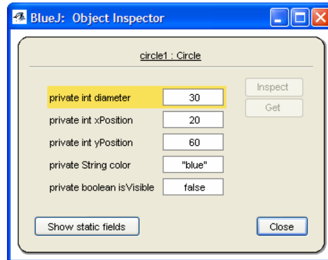
- to initialize, must call the object's constructor: `OBJECT = CLASS(`*`PARAMETERS`*`);`
- to call a method: `OBJECT.METHOD(`*`PARAMETERS`*`)`

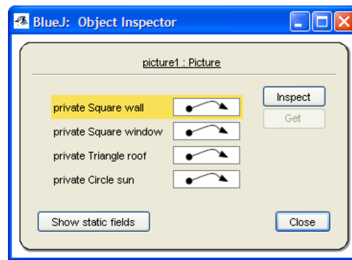
```
Circle circle1;                Die d8;
circle1 = new Circle();        d8 = new Die(8);
circle1.changeColor("red");    System.out.println( d8.roll() );
```

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





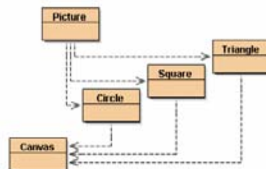
- of course, you can further inspect the contents of object fields

we will consider the implications of primitives vs. objects later

Picture example

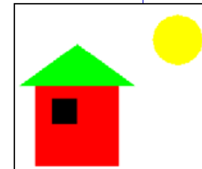
recall the Picture class, whose `draw` method automated the process of drawing the picture

- 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();
    }
}
```



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
- 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

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, maximum step size)
access the dot's position
take a step
reset the dot back to the beginning
display the dot's color & position

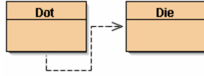
once the `Dot` class is defined, a `DotRace` will be much simpler

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

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

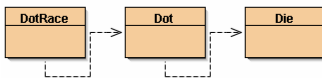
CREATE AND PLAY

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

    public Dot(String color, int maxStep)
    {
        die = new Die(maxStep);
        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);
    }
}
```


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(int maxStep)
    {
        redDot = new Dot("red", maxStep);
        blueDot = new Dot("blue", maxStep);
    }

    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();
    }
}
```

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

    public int getGoalDistance()
    {
        return goalDistance;
    }

    . . .
```


Adding a finish line

`step` method needs a 3-way conditional:

- either blue crossed or red crossed or the race is still going on

```
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();
    }
}
```

ugly! we want to avoid duplicate code

fortunately, Java provides *logical operators* for just 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

Adding a finish line

here, could use `||` to avoid duplication

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

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


Further changes

EXERCISE: make these modifications to your `DotRace` class

- add the `goalDistance` field
- modify the constructor to include the goal distance
- add an accessor method for viewing the goal distance
- add an if statement to the step method to recognize the end of the race

what if we wanted to display the dot race visually?

- could utilize the `Circle` class to draw the dots
- unfortunately, `Circle` only has methods for relative movements

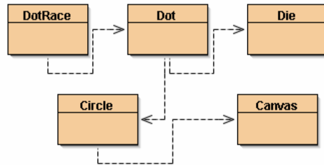
  we need a `Circle` method for absolute movement (based on a Dot's position)

```
/**
 * Move the circle to a specific location on the canvas.
 *   @param xpos the new x-coordinate for the circle
 *   @param ypos the new y-coordinate for the circle
 */
public void moveTo(int xpos, int ypos)
{
    erase();
    xPosition = xpos;
    yPosition = ypos;
    draw();
}
```

Adding graphics

due to our modular design, changing the display is easy

- each Dot object will maintains and display its own `Circle` image



- add `Circle` field
- constructor creates the `Circle` and sets its color
- `showPosition` moves the `Circle` (instead of displaying text)

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

    public Dot(String color, int maxStep)
    {
        die = new Die(maxStep);
        dotColor = color;
        dotPosition= 0;
        dotImage = new Circle();
        dotImage.changeColor(color);
    }

    public int getPosition()
    {
          return dotPosition;
    }

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

    public void reset()
    {
        dotPosition = 0;
    }

    public void showPosition()
    {
        dotImage.moveTo(dotPosition, 30);
        dotImage.makeVisible();
    }
}
```


Graphical display

note: no modifications are necessary in the `DotRace` class!!!

- this shows the benefit of modularity
- not only is modular code easier to write, it is easier to change/maintain
- can isolate changes/updates to the class/object in question
- to any other interacting classes, the methods look the same

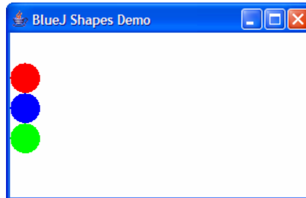
EXERCISE: make these modifications to the `Dot` class

- add `Circle` field
- create and change color in constructor
- modify `showPosition` to move and display the circle

Better graphics

the graphical display is better than text, but still primitive

- dots are drawn on top of each other (same yPositions)
- would be nicer to have the dots aligned vertically (different yPositions)



PROBLEM: each dot maintains its own state & displays itself

- thus, each dot will need to know what yPosition it should have
- but yPosition depends on what order the dots are created in DotRace
  (e.g., 1st dot has yPosition = 30, 2nd dot has yPosition = 60, …)

- *how do we create dots with different yPositions?*

Option 1: Dot parameters

we could alter the Dot class constructor

- takes an additional int that specifies the dot number
- the dotNumber can be used to determine a unique yPosition
- in DotRace, must pass in the number when creating each of the dots

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

    public Dot(String color, int maxStep, int num)
    {
        die = new Die(maxStep);
        dotColor = color;
        dotPosition= 0;
        dotImage = new Circle();
        dotImage.changeColor(color);
        dotNumber = num;
    }

    . . .

    public void showPosition()
    {
        dotImage.moveTo(dotPosition, 30*dotNumber);
        dotImage.makeVisible();
    }
}
```

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

    public DotRace(int maxStep)
    {
        redDot = new Dot("red", maxStep, 1);
        blueDot = new Dot("blue", maxStep, 2);
        greenDot = new Dot("green", maxStep, 3);
    }

    . . .
}
```

this works, but is inelegant

- why should DotRace have to worry about dot numbers?
- the Dot class should be responsible

Option 2: a static field

better solution: have each dot keep track of its own number

- this requires a new dot to know how many dots have already been created
- this can be accomplished in Java via a *static field*

```
private static TYPE FIELD = VALUE;
```

- such a declaration creates and initializes a field that is shared by all objects of the class
- when the first object of that class is created, the field is initialized via the assignment
- subsequent objects simply access the existing field

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

    private static int nextAvailable = 1;
    private int dotNumber;

   public Dot(String color, int maxStep)
   {
        die = new Die(maxStep);
        dotColor = color;
        dotPosition= 0;
        dotImage = new Circle();
        dotImage.changeColor(color);

        dotNumber = nextAvailable;
        nextAvailable++;
   }

   . . .

   public void showPosition()
   {
       dotImage.moveTo(dotPosition, 30*dotNumber);
       dotImage.makeVisible();
   }
}
```

try it! | how do static fields appear when you inspect? | could `die` be static?

Class/object summary

a class defines the content and behavior of a new type

- *fields*: variables that maintain the state of an object of that class
  fields persist as long as the object exists, accessible to all methods
  if want a single field to be shared by all objects in a class, declare it to be *static*

  to store a primitive value: declare a variable and assign it a value
  to store an object: declare a variable, call a constructor and assign to the variable

- *methods*: collections of statements that implement behaviors
  methods will usually access and/or update the fields to produce behaviors
  statement types so far: assignment, println, return, if, if-else, method call (internal & external)

  parameters are variables that store values passed to a method (allow for generality)
  - parameters persist only while the method executes, accessible only to the method

  local variables are variables that store temporary values within a method
  - local variables exist from point they are declared to the end of method execution

- *constructors:* methods (same name as class) that create and initialize an object
  a constructor assigns initial values to the fields of the object (can have more than one)

Tuesday: 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)