Computer Science 120
Introduction to Programming
Siena College
Spring 2011

# Topic Notes: Defining Classes

## Numeric Data Types

Before we move on to our next major topic, a few words about how Java represents numbers.

Java includes four commonly used numeric types:

- `int`,
- `long`,
- `float`
- `double`

(There are others, but we won't discuss them here.)

The types `int` and `long` both represent integers. Type `int` represents numbers between about $-2 * 10^9$ and $2 * 10^9$, while `long` represents integers up to about $10^{19}$.

Numbers written with decimal points are represented by the types `float` and `double`. The type `float` only retains about 7 digits of precision, so we usually use `double` instead. Type `double` has about 15 digits of precision and can represent numbers up to about $10^{308}$ and as small as $10^{-308}$. These numbers can be written either simply with a decimal point, e.g., $4.735$, or in scientific notation, *e.g.*, $5.146$E+47, representing $5.146 * 10^{47}$.

Because of the way many of the Java libraries are written we will tend to use type `int` for integers and `double` for numbers with decimal points. Occasionally we will use `long` when we need to represent more digits with integers, but we will find no need to use `float` in this course.

Like `int`, the type `double` supports operations +, -, *, and /, but does not have an operation corresponding to %. The results of applying these operators to `double`s is an answer which is also a `double`. Thus 3.0 / 2.0 = 1.5.

If an arithmetic operator has one operand with type `int` and the other with type `double`, the result will be a `double`. Basically what happens is that Java recognizes that the two operands are of different types (which it is not happy about), and thus attempts to make them be of the same type. The simplest thing, from Java's point of view, is to convert `int`s to `double`s, as no loss of precision results. As we saw earlier, if both operands are of type `int`, the result will also be of type `int`.

One must be careful in performing divisions to be aware of the types of the operands, as the results may differ depending on their types. *e.g.*, 3 / 2 = 1, while 3.0 / 2.0 = 3.0 / 2 = 3 / 2.0 = 1.5.

## Defining Classes

So far, we have been operating directly on the objectdraw graphics primitives such as `FramedRect` and `FilledOval`. Your work on the second part of the laundry lab may have started to give you an indication that such an approach can become tedious as our programs become more complex. You had to update both a `FramedRect` and a `FilledRect` that, together, form the swatch of laundry during the dragging operations.

Suppose we wanted to augment our basketball game to have a more realistic-looking basketball:

See Example: NiceBasketball

This is a much nicer-looking basketball than what we had in our original game. We do have one new objectdraw construct to help us out here: the `FramedArc`. This draws only a part of a `FramedOval`. It determines which part with two additional parameters, a start angle and an arc angle, both specified in degrees. The start angle refers to the angle from the right edge of the oval, counterclockwise, where we are to start drawing. The arc angle refers to the number of degrees, counterclockwise, to draw.

We will not focus much on that aspect of this example, but since `FramedArc`s are useful building blocks (along with their cousins the `FilledArc`s), another example demonstrates a bit more about how to use them:

See Example: Arcs

But back to our nice basketball. Consider how messy our code would become if we decided to add a dragging functionality to this basketball. Not just one, not just two, but six objects would need to be moved in the dragging code.

That certainly would become tedious. And we might also want to use this nice basketball in other programs and in other ways.

Java provides a mechanism that will help us to define a `class` of objects, much like objectdraw defined our graphics primitives, that we can use in our programs in a very convenient way.

Consider this example, which defines a class called `NiceBBall` in `NiceBBall.java` and then makes use of it in an updated version of our basketball game in a familiar `WindowController` class called `FancyBasketball`.

See Example: FancyBasketball

What is a `NiceBBall`? As we can see, it is a basketball. The ability to draw basketballs is not a standard feature of Java or of the objectdraw library. This program shows how Java allows us to define new classes of objects appropriate to the needs of the particular program we are writing.

You've already been defining classes – the `WindowController` extension that has been the framework for each program we've seen is a class definition. From this you are familiar with the basics of the structure that will be required to define a `NiceBBall` class.

```
public class Name
{
    constant and variable declarations

    methods
}
```

Our earlier classes always extended `WindowController`, which indicated that they would be designed to respond to mouse activities in the window. This will not be the role of the `NiceBBall` class, so it will not `extend WindowController` and it will not include methods like `onMouseClick` or `onMouseDrag`. Instead, the body of the `NiceBBall` class will consist of definitions of methods corresponding to the things we want to be able to tell a `NiceBBall` to do, like `move` and `moveTo`.

When we define such a class:

1. We declare instance variables describe the parts and state of an object of that class. Any individual basketball is composed of ovals, lines and arcs so the instance variable of the `NiceBBall` class refer to these objects.
2. We provide a special method called a *constructor*. We've been using constructors for our library objects - whenever we write a construction. When we say "`new FramedRect`", Java knows what to do to make a `FramedRect` appear on the screen because someone defined a constructor for `FramedRect`. When we write a constructor, we will write a list of statement to construct the components of the object desired (the parts of the basketball) and associate the with instance variables defined in the new class.
3. We define methods - *i.e.*, lists of statement explaining how to perform the actions we want the new objects to know how to do.

---

## Mutator Methods

Let's first consider one of the methods in the `NiceBBall` class:

```
public void move(double dx, double dy) {
  body.move(dx, dy);
  // ...
  // and all of the other parts also are moved
}
```

The first keyword in a method declaration is either `public` or `private`. (The keyword `protected` is also allowed, but we will not use it in this course.) These keywords determine the "visibility" of a method. If a method is declared to be `public` then it can be called from methods in other classes. For example, the `onMouseDrag` method of the `FancyBasketball` class calls the `move` method to the `NiceBBall` object we create. This would not be possible if that first keyword had been `private`.

All of our instance variables are `private`, because they should only be used inside of the class where they are declared. Occasionally we will have constants that need to be visible outside of the class that contains them. In that case we will declare them as `public static final ...`

The next keyword in the declaration of `move` above is `void`. This (not very intuitively) indicates that it is a mutator method. That is, it simply performs an action rather than returning a value which can be used in an expression.

Next comes the method's name, `move` for example. After this we place formal parameter declarations in parentheses.

Most of the methods we have define so far, have expected one piece of information when invoked (*e.g.*, the `Location` provided to mouse handing methods). Therefore, the headers of these methods have all declared a single formal parameter "`Location point`".

The methods associated with an object like a `NiceBBall` may expect to be provided other types of information when invoked. For example, the `move` method will expect a pair of numbers specifying x and y offsets. The parameters listed in a method's header must correspond to the information that will be provided when the method is invoked.

The parameters of move are of type `double`. These give the distance that the object should move in the x and y directions, respectively. Recall that the way we use the `move` method is as follows:

```
    anObject.move(10, 20);
```

In method headers/signatures, parameters serve as declarations of variables. Thus the occurrence of "`double dx`" between parentheses serves to declare the variable `dx` as a parameter of type `double`. Parameters are used to pass along information to a method. Thus the body of method `move` can use the variable `dx` to represent the number provided when the method was invoked to specify the desired x offset. The correspondence between the formal parameter names and the actual parameter values is determined by their order. The first actual listed in a method invocation is associated with the first formal parameter listed in the header and so on. So in `FancyBasketball`'s `onMouseDrag` method, when the call

```
     ball.move(point.getX() - lastMouse.getX(),
               point.getY() - lastMouse.getY());
```

is executed, Java transfers control to the `move` method of `NiceBBall` and initializes `dx` and `dy` to the results of `point.getX() - lastMouse.getX()` and `point.getY() - lastMouse.getY()`, respectively.

The `NiceBBall` class has two other mutator methods: `moveTo` and `changeToRandomColor`. The method header for `moveTo` is very similar to that of `move`. The statement list found in `moveTo`'s method body is much shorter than that `move`'s, however. This is because the definition of moveTo takes advantage of the definition of move: moveTo computes the offset from the ball's current location to the desired location and then invokes the move method to move the 6 pieces of the ball.

The changeToRandomColor mutator method is included mainly to emphasize that the methods we define in a class do not need to be only those we have seen from the graphics primitive we've been using so far. Many of our methods will have familiar names like move, contains, or removeFromCanvas, but that's only because we are often defining graphical objects and those are natural operations on graphical objects. In changeToRandomColor, we have provided functionality for our NiceBBall objects that does not exist for standard graphics primitives like FramedRect.

---

## Constructors

Constructors are used to perform the actions which must be undertaken when the object is created. As a result, they often perform the same kind of actions as the begin method in the classes extending WindowController. In particular, they typically provide initial values of instance variables and create the graphic objects needed in a class.

The form of the constructor will be very similar to that of methods:

```
    public ClassName( params here ... )
```

The name of the constructor will always be the same as the name of the class being defined. Thus if we were defining a class named NiceBBall, the constructor would have the same name. Constructors are usually public, and may have as many parameters as are necessary to provide them with the information necessary to initialize instance variables. However, constructors differ from methods by omitting the void before the constructor's name.

As an example, the constructor for class NiceBBall has parameters corresponding to the starting location of the upper left hand corner of the ball, the size of the ball desired and information on what canvas the ball should be drawn. Thus its declaration looks like:

```
  public NiceBBall(double left, double top,
                   double size, DrawingCanvas aCanvas)
```

The type of the last parameter of the NiceBBall constructor is new to us. DrawingCanvas is the type of the variable canvas that we have been using when creating graphic objects. It is the type of a surface that can be used for drawing graphic objects. To this point, we have only been using the standard one (called canvas) that is defined for us by the WindowController. Since NiceBBall does not extend WindowController, it would not have any idea what canvas is unless we tell it, and we tell it that information by passing it as a parameter to the constructor.

---

## Accessor Methods

The last piece of the definition of the `NiceBBall` class is the specification of its *accessor methods*: `contains`, `getX`, and `getY`. Accessor methods allow us to ask questions of an object and get information back.

The `contains` method's header looks like:

```
public boolean contains(Location point)
```

This differs from the mutator methods in that the magic word `void` has been replaced by the name of the type `boolean`. This is because the method we are defining here is an accessor method that will return a boolean value. That is, the results of sending this message to (*i.e.*, calling this method of) an object will be a value: either `true` or `false`. As a result, it is used in a context expecting a `boolean` result, such as:

```
if (anObject.contains(lastPoint)) ...
```

In fact, the word `void` serves a similar function. It tells us what kind of value the method being declared will produce – in the case of a mutator method: no value.

---

## Variables and Scope

The previous example also demonstrates another way to declare variables. We have seen instance variables, named constants, and formal parameters. The methods `move` and `changeToRandomColor` each have *local variables*.

These are variables declared right inside a method, and exist only in that method's body. The declarations look a lot like an instance variable declaration, except that we omit the "`private`" qualifier, since local variables are already very "private" – the name is only meaningful within the method where it is declared.

To summarize the ways we have to declare names for variables and constants:

- Instance variables are declared outside of any method and are visible inside all methods. The value of an instance variable is retained from one method call to the next.
- Named constants are also declared outside of any method and are visible inside all methods. The value never changes once set initially.
- Formal parameters are used to communicate information to a method from its caller. The names are meaningful only within the method. New values are provided (by the caller in the corresponding actual parameters) each time the method is called.
- Local variables are used to store temporary values needed within the execution of a method. They exist only within the method in which they are declared and do not retain their values from one call of the method to the next.

Beginning Java programmers are sometimes confused about when to use an instance variable and when to use a local variable. The correct choice depends on how long the information that will be stored in the variable needs to exist. If it only needs to exist within a single method execution, it should be a local variable. If it needs to exist across method calls, it should be an instance variable.

---

## Multiple Instances of a Custom Class

One of the great advantages of defining a custom class is that we can then instantiate as many objects of that class as we wish.

For example, we could update our "mouse droppings" program to draw lots of nice basketballs instead of little red circles:

See Example: DrawBBalls

For another example, let's return to our laundry theme. We'll construct a somewhat more believeable laundry item that looks like a T-shirt, create two instances, and then drag them around the screen.

See Example: Drag2Shirts