Computer Science 211
Data Structures
Mount Holyoke College
Fall 2009

Topic Notes: Java and Object-Oriented Programming Review

There are many ways to write a program to solve a particular problem correctly. The ways to write it so as to run *efficiently* are much more limited.

In addition to efficiency and correctness, we will think about these implmentation goals for our programs:

1. Robustness
    - produce correct output for all inputs - including erroneous input
2. Adaptability
    - a program can evolve over time with new requirements
3. Reusability
    - develop general-purpose code that may be used in multiple situations

---

Programming Languages

We will be programming in Java.

There are many programming languages, and Java is just one example.

We'll take a look at a few others, just for fun. These are all "Hello, world" programs.

See Example:
`/home/jteresco/shared/cs211/examples/Hello`

See `http://www2.latech.edu/~acm/helloworld` for many more examples.

These languages all basically do the same thing, though some are much better than others at certain types of tasks. We choose to study data structures and advanced programming in Java because it is a modern, object-oriented language that runs on all modern computer systems. An appropriate choice of programming language makes it easier to write high-quality software.

All computer languages are an *abstraction* to make it more convenient to get computers to do what we want them to do. We could write in 0's and 1's, but a variety of languages have been developed to facilitate the development of software.

Most langauages, including C and C++ have compilers that translate source code into a executable program that runs on a particular machine. Java is different. All Java compilers translate to a particular *virtual machine*, which, in turn, runs on specific computers.

This gives Java some advantages that we will discuss along the way. For now, we will look at programming in Java as a general-purpose modern object-oriented programming language.

Java Review/Overview

Back to our Hello, world example.

Things to note in Hello.java:

- Everything in Java has to be in a class. More about classes in a minute. In C, there are no classes and in C++ there are classes, but you can write functions outside of any class in addition to class methods.
- Many of you have have seen only Java applets – this is a Java *application*. We will look primarily at applications here.
- Each class can have methods. An application has a class that must have a `main` method with the method signature:

  ```
  public static void main(String[] args)
  ```

  Exactly what is meant by all of these will become clear later, but basically this is where execution will start when we run this program.
- To execute this program using the command-line (text) interface to Java:

  ```
  javac Hello.java
  ```

  This *compiles* the Java program (in the `.java` file) into a `.class` file. A `.class` file contains *Java Btyecode* that is ready to be executed in a JVM.

  To run it:

  ```
  java Hello
  ```

  The JVM is what allows Java to be portable. We can compile the `.java` file to a `.class` file on any computer and any other computer running the same version of Java will be able to execute it.
- Comments:

  ```
  // starts comments which go to end of line
  /* multi-line comments
        are done like this */
  ```

You most certainly have been strongly encouraged or, more likely, required to document your program using comments in your previous courses.

The comment above the header in `Hello.java` is a special kind of comment, it starts with `/**` indicating that it is a *Javadoc* comment. These comments are used to generate documentation for Java programs automatically.

We can generate the Javadoc for this simple program:

```
javadoc Hello.java
```

And then we can view it in our favorite web browser.

Note that the "class comment" is included at the top, the `@author` tag is ignored but useful.

The comment is in HTML (note the `<P>` tag to mark a new paragraph).

Each method is listed, and the method comment (also beginning with `/**`) includes a description and the parameters with the `@param` tag.

We will see other Javadoc tags in later examples.

- The text output is made by a call to `System.out.println()` which takes one argument – a string to print to the terminal.

  This plays the role of C's `printf` and C++'s `cout`.

  Note: `System.out.print()` does the same, but without the new line at the end.

Object-Oriented (OO) Design

- Objects are building blocks of software systems
- Program is collection of interacting objects
- Objects cooperate to complete a task
- Objects communicate by sending messages to each other

Objects can model objects from world:

- Physical things (car, student, card, deck of cards)
- Concepts (meeting, date)
- Processes (Sorting, simulations)

Objects have:

- Properties (blue, Ford, 4 doors)
- Capabilities (drive, change speed, admit passenger, brake)

Grocery item example: some properties we will consider are item size and item price and unit price. We will change one property - price.

Objects are responsible for knowing how to perform actions:

- Commands: change object's properties (e.g., set speed, change price)
- Queries: Provide answer based on object's properties (e.g., how fast?, what is the size?)

Capabilities implemented as *methods*.

- invoked by sending messages to an object
- *mutator* methods change an object's properties
- *accessor* methods query an object's state

Properties of an object are implemented as *fields* or *instance variables*.

- constitute the "state" of the object
- the state affects how an object reacts to messages

Properties can be:

- attributes - descriptions (e.g., red)
- components (e.g., doors)
- associations (e.g., driver)

Let's consider an example a bit different from the ones in the text in Chapter 1 (which I encourage you to look at as well).

Suppose our goal is comparison shopping. Among a collection of items in a store, each of which has a different price and size, which is the best deal?

To attempt to design this in an object-oriented fashion, we will create objects corresponding to the items we will compare.

We will develop an object to hold an item which has an associated price and size. Our class will *encapsulate* this data, meaning it can only be accessed by the methods of our class.

See Example:
`/home/jteresco/shared/cs211/examples/UnitPrice`

Our class to represent one of these items is `PricedItem`.

What are the capabilities of a PricedItem?

- item creation – need initial price and size
- item deletion
- query item price, size, unit price
- set item price

What are the properties?

- price
- size
- unit price?

Let's look at the class:

We have instance variables to hold the state of the item – `price` and `size`, both as `double`s.

We do not include the unit price as an instance variable, since we can compute it easily from the price and size.

This is an important design decision – what information to keep and what information can be easily recomputed. Here, any two have enough information so we will just keep `price` and `size`.

If we decided to keep all three, it would make some of our accessor methods simpler, but it would complicate out mutators since they would need to update multiple values to keep the state of the class consistent.

So we declare

```
protected double price;
protected double size;
```

These `protected` variables can be accessed from all methods in the class and their value is persistent through the life of the object. These should only be used for things that need this persistence.

We can declare variables and methods:

- `public`: Everyone has access to this feature

- `protected`: Features are accessible within methods of the class (or extensions), but not accessible outside.

  We will generally declare instance variables as `protected`.

  If we declared them as `public`, then another class could access them (or worse yet, modify them) without going through our class methods.

- `private`: Features not visible in extensions or outside of this object object.

  `protected` and `private` declarations are used to hide implementation details from clients.

  This makes it easier to change our implementation later if needed, without affecting other classes that make use of our class.

Each instance variable needs a type. They can be one of Java's primitive types:

- `boolean`
- `char` (16 bit)
- `int` (32 bit)
- `long` (64 bit)
- `float` (32 bit)
- `double` (64 bit)

They can also be other objects. We will see examples of this shortly.

Next, we need one or more *constructors*. To construct our item, we will need to specify the price and the size as arguments to the constructor.

```
public PricedItem(double price, double size) {
    this.price = price;
    this.size = size;
}
```

Note that a constructor has the same name as the class, must be declared as `public`, and must not have a return type.

This constructor has two parameters. All parameters in Java are passed by value. That is, any assignment to a parameter inside a method body is forgotten when the method returns. They are basically local variables that are initialized to the value passed in the actual parameter by the caller.

Note the use of `this.` to denote the instance variables with the same name as the formal parameters. `this` is assumed and is not needed unless there is a conflict such as in our example. Without `this` in this case, we would write `price = price` and just set the value of the formal parameter variable to itself (not especially useful).

Keyword `this` can be used in a method to refer to the object executing the method (i.e., the receiver of the message).

An object can call one of its own methods by writing `this.m(...)` or just `m(...)` (the `this` is assumed).

More interestingly, we can pass `this` as a parameter to another object. We will see examples later.

Next, an accessor method for the unit price.

```
public double unitPrice() {
    return price/size;
}
```

Note here that the method is declared `public` so it can be called from outside the class, and the `return` statement. We alternately could have declared a local variable to store the computed unit price and return that, but it is not necessary.

Finally, a mutator method to change the price of an item:

```
public void setPrice(double newPrice) {
    price = newPrice;
}
```

We save the new price in our instance variable. That's all. Note that here, the formal parameter name is different, so `this.` is not necessary.

We also include a `toString()` method which allows instances of the class to print out their own information in a meaningful way.

We have a short driver program to make use of our `PricedItem`s in `UnitPriceThree.java`.

This is another Java application, i.e. a class with a `main` method.

We create three items to compare, plus an extra variable to refer to the one we determine to be the best deal. These are local variables, accessible only within the `main` method. No `private`/`protected`/`publi` qualifiers are needed or allowed on local variables.

We first construct the three items. `new` creates a new instance of a class.

Next, we call a *private helper method* to find the item among our three that has the lowest price. This could be done without the helper method, but since we call it twice, it makes sense to place that code into the helper method.

We're calling a method of (or sending a message to) ourselves.

We'll worry about the actual method definition in a minute. For now, let's look at the printout.

We're printing out a `String`. Note that we are performing a `+` operation of a string constant and an object.

A quick aside about Java `String`s.

- We can define string constants: `"This is a string"`
- The `+` operator is used for concatenation: `"This is "+"a string."`
- The results of a string + anything is string: `"Chapter "+ 2` is the same as `"Chapter 2"`
- `String`s are not a base type, but they're a bit unusual for some reasons we will see later.

When we try to use an object reference as a `String`, the object's `toString()` method is called.

All objects come with a default `toString` method, but we have written our own for `PricedItem`s, that print out some meaningful information about the object.

We then change the price of one of the items and see if that changes the best deal.

In our helper method, we have two local variables. We call methods of the `PricedItem`s passed in to get their unit prices.

Big disadvantage of this version: we're stuck with just three items. If we want more, we need to declare more variables, change the number of parameters to the helper method, change the implementation of the helper method, each time we change the number of items.

A second driver program, `UnitPrice.java`, uses arrays.

We declare the array, here as a local variable. Note the square brackets on the declaration to indicate that we will have an array of `PricedItem` objects.

Note: We never include the size of the array in the declaration

```
int[10] scores // Illegal in Java!
```

An array declaration in Java does not create space for the array itself. We need to construct the array.

Array construction does not construct objects, it only creates the array where we can store references to `PricedItem`s. These are initialized to `null` – a reference to nothing. We still need to construct the actual objects in the arrays.

Our helper method now also uses the array. The use of the array should look familiar to everyone.

Arrays are numbered starting at 0, and indices go up to `array.length`-1.