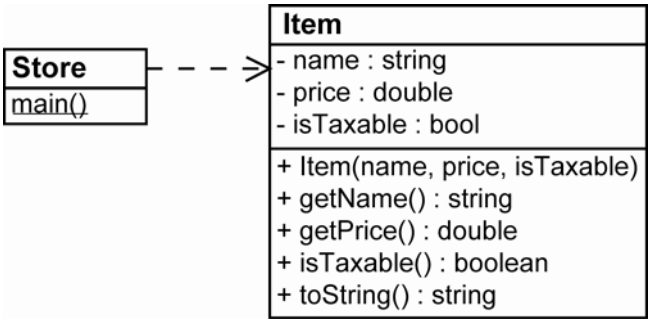
Chapter 7 – Classes & Objects, Part C

These notes focus on static variables and methods, a few common errors, and anonymous objects.

Example

1. A store would like to have a program that would allow the user to input information about the items they sell. The store sells many items and each item has a name, a price, and an indicator as to whether the item is taxable or not. For now, that's all we know. We'll learn more about the program later. So, let's get started designing the solution using object-oriented programming.

2. When we read a problem, we look for *nouns*. Many times, nouns represent either a class or an attribute. Above, we see that *store, item*, *name*, and *price* are all nouns. A little bit of thought tells us that *name* and *price* are *attributes* of the *Item* class. So, we'll need an *Item* class. But what about that indicator for taxation? Sounds like another *attribute* of an *Item*. Next, it looks we should write a program in main() that uses the *Item* class. Perhaps we could name that class, *Store*.



3. Writing the *Item* class should be straight forward. We have three read-only properties to code getters for. For the toString() method, we'll return a string that uses all the getters. Thus, by calling toString(), we'll be testing all the methods.

```
public String toString()
{
      String item = String.format(

            "Product: %s, Price: %.2f, Taxable: %b",

            getName(), getPrice(), isTaxable() );

      return item;
}
```

4. At this point, we need to test what we have written. So, let's just create some items in the store and iterate (loop) over them printing each item. Since we are likely to have a lot of items, we should probably put them into an array. Consider this code in main():

```
public class Store
{
     public static void main(String[] args)
     {
        Item[] items = createItems();

        for( Item i : items ) System.out.println( i.toString() );
     }

     private static Item[] createItems()
     {
        Item[] items = new Item[5];

        items[0] = new Item("grits", 3.00, false);
        items[1] = new Item("bread", 2.50, false);
        items[2] = new Item("soda",  6.00, true);
        items[3] = new Item("newspaper", 1.50, true);
        items[4] = new Item("eggplant", 3.50, false);

        return items;
     }
}
```

5. Suppose that we want to calculate the total price of all the items where there is a 10% rate on taxable items.

```
for( Item i : items )
{
     total += i.isTaxable() ? i.getPrice()*1.1 : i.getPrice();
}

System.out.printf( "The total is $%.2f\n", total );
```

6. Suppose that we want to display all the taxable items that cost more than $3? The technique we use is to simply iterate over the items and check which ones meet the criteria. We call this process *filtering*.

```
for( Item i : items )
{
     if ( i.isTaxable() && (i.getPrice() > 3.0 ) )

          System.out.println( i.toString() );
}
```

Static Methods

1. Suppose you anticipate the need to be able to find the minimum priced item in the *array of items*. First, should such a method be a static method in main() (*i.e.* a part of the application) or a method in the Item class (*i.e.* a behavior of the *Item* class)? We answer that question by determining if this requirement is more of a requirement of the application, something that could change as opposed to a *natural* behavior/responsibility of the *Item* class, something that is unlikely to change. In the former, we may make a static method in main(). In the latter, we would probably make a method in the class. So, we probably agree that it wouldn't be unreasonable to have the Item class supply a method to find the item with the minimum price, given an array of Items.

   However, this behavior is a bit different. A behavior that pertains to an individual object is implemented as a public method. The method described above pertains to an array of items. So, it doesn't make sense to make it a public method. We would generally make such a method a *static method* (also called a *class method*). A *static* method does not belong to any given instance (object); it belongs to the class itself. Thus, we use the class name to call a *static* method:

```
Item minItem = Item.minPrice( items );
```

   Next, we write the *minPrice* static method in the *Item* class. It will accept an array of *Item*'s and return an *Item*, the item with the lowest price and it will.

```
public static Item minPrice( Item[] items )
{
      Item minItem = items[0];

      for( Item i : items )

            if ( i.getPrice() < minItem.getPrice())

                  minItem = i;


      return minItem;
}
```

   Which we can test with:

```
Item[] items = createItems();

Item minItem = Item.minPrice( items );

System.out.printf( "The minimum priced item is:\n%s",

                        minItem.toString() );
```

Static Variables

Suppose that we want to keep track of the total number of *Item*'s that have been created. We could certainly keep track of this in main(), but another idea is to make it a *static property* (*static variable*) of the *Item* class. Such a variable can be *shared* among the instances. The values of static variables are stored in a common location that all objects have access to. We'll add this variable to the Item class:

```
public static int numCreated = 0;
```

And, every time the *Item* constructor is called, we'll increment *numCreated*:

```
public Item( String name, double price, boolean isTaxable )
{
      this.name = name;
      this.price = price;
      this.isTaxable = isTaxable;

      numCreated++;
}
```

Finally, to test this variable, we can create 5 items, destroy them, then create 5 more:

```
      Item[] items = createItems();
             items = createItems();

      System.out.printf(
            "The total number of Items created is %d\n",
            Item.numCreated );
```

Initialization and Common Errors

1. As we have seen, a class can have data fields that are primitive data type or reference data types. Primitive data types take on the default values of 0 for numeric types and false for boolean. Reference data types take on, by default, the special literal value, *null*. It indicates that there is no reference contained in the variable; it points nowhere!

   For instance, suppose we wanted an *Order* class, where each order contains a bunch of items. For simplicity, we will assume that an order has just one item, to illustrate a very common programming error. The *Order* class looks like this:

```
class Order
{
     private Item item;

     public Order() {}

     public void addItem( Item i )
     {
        item = i;
     }

     public double getPrice()
     {
        return item.getPrice();
     }
}
```

   Now, consider two different errors that can occur in your main() code. This is a compile error, forgetting to create an object before using it in main():

```
        Order o;

        double price = o.getPrice();
```

   The resulting compile error is:

```
Store.java:41: variable o might not have been initialized
    double price = o.getPrice();
                   ^
```

   In other words, *there is no object, o*. However, detecting such a situation *inside* a class is impossible because even though when you create an object, you might not initialize/create a reference data type, you still might have a method that is designed to do just such a thing. For instance, in the *Order* class, when we create an order, no item is created, thus, the *Order*'s data field *item* is *null*. However, the compiler does not know whether you have called *addItem* or not. Thus, code in main() as shown below, will **not** be caught at compile time, but will generate a **run-time error**:

```
        Order o = new Order();

        double price = o.getPrice();
```

The resulting error is called a ***NullPointerException***. Though in this example, and especially since we are explaining it, the error seems obvious, I promise you that this error will *haunt* you the rest of your programming days. This is an example of the error:

```
Exception in thread "main" java.lang.NullPointerException
  at Order.getPrice(Store.java:126)
  at Store.main(Store.java:41)
```

Note that in this example, line 126 is the place where java first encountered a *null* pointer (*i.e.* a reference to an object that has not been created).

```
return item.getPrice();
```

Sometimes that is helpful, but it doesn't tell you *why* the object is not created yet. The same kind of thing holds true for arrays and strings and other reference types used inside a class.

Finally, to use the object correctly, we must write code like this:

```
Order o = new Order();

Item i = new Item("grits", 3.00, false);

o.addItem(i);

double price = o.getPrice();
```

Annonymous Objects

1. Most of the time, we create an object and assign it to a variable. Thus, the object has a reference, a name inside the program. Occasionally, we find use for an *anonymous object*, an object without any reference to it. This is an example, written in two different ways:

```
System.out.println( new Item("grits", 3.00, false).toString() );
System.out.println( ( new Item("grits", 3.00, false) ).toString() );
```