SIENAcollege
Computer Science

Computer Science 225
Advanced Programming
Siena College
Spring 2020

Topic Notes: More Java: Odds and Ends

This final set of topic notes gathers together various odds and ends about Java that we did not get to earlier.

Enumerated Types

As experienced BlueJ users, you have probably seen but paid little attention to the options to create things other than standard Java classes when you click the "New Class" button.

One of those options is to create an `enum`, which is an *enumerated type* in Java.

If you choose it, and create one of these things using the name `AnEnum`, the initial code you would see looks like this:

```
/**
 * Enumeration class AnEnum - write a description of the enum class here
 *
 * @author (your name here)
 * @version (version number or date here)
 */
public enum AnEnum
{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
```

So we see here there's something else besides a class, abstract class, or interface that we can put into a Java file: an `enum`.

Its contents are very simple: just a list of identifiers, written in all caps like named constants. In this case, they represent the days of the week. If we include this file in our projects, we would be able to use the values `AnEnum.MONDAY`, `AnEnum.TUESDAY`, ... in our programs as values of type `AnEnum`. Maybe a better name would have been `DayOfWeek`..

Why do this? Well, we sometimes find ourselves defining a set of names for numbers to represent some set of related values. A programmer might have accomplished what we see above by writing:

```
public class DayOfWeek {

    public static final int MONDAY = 0;
    public static final int TUESDAY = 1;
    public static final int WEDNESDAY = 2;
    public static final int THURSDAY = 3;
    public static final int FRIDAY = 4;
    public static final int SATURDAY = 5;
    public static final int SUNDAY = 6;
}
```

And other classes could use `DayOfWeek.MONDAY`, `DayOfWeek.TUESDAY`, etc., but would have to store them in `int` variables.

By using an `enum`, we know that values of that type could only be the ones we specified.

This example and more are on the Java tutorials page about Enum Types:

`https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html`

Note in particular that an `enum` can be more than just a number or a string. We can define a constructor and methods.

The `final` Keyword

You have most likely used the `final` keyword in Java when defining constants:

```
  private static final int BOARD_SIZE = 100;
```

or local to a method:

```
  final String WIN_MESSAGE = "You win!";
```

In this context, `final` indicates that the value of the name being defined cannot be changed subsequently. A statement that tries to assign a new value to such a name will trigger a compile error.

You are less likely to have encountered the `final` keyword as a qualifier on a formal parameter:

```
   public void move(final double dx, final double dy) {
     ...
   }
```

Normally, you can use parameters as local variables and change them as you see fit in the body of the method, but adding the `final` qualifier disallows this.

Note however, that this only means that the variable cannot be changed. It does not mean that the variable's value cannot be used to change something. For example, consider this method:

```
public void f(final ArrayList<Integer> a) {

     // this would be an error:
     // a = new ArrayList<Integer>();

     // but this is permitted:
     a.add(27);
}
```

Even though `a` is `final`, and it cannot be modified to refer to a different `ArrayList`, there is nothing stopping that reference from being used to modify the `ArrayList` to which it points.

Advanced Inheritance Concepts

We've made use of inheritance repeatedly, but we now consider some issues that can come up that are a bit more advanced.

Access Control with Overriding

What should we do about the access control of a method in a derived class that overrides a method in its base class? That is, if the method was `public` in the base class, must it be `public` in the derived class? Yes, in that case, but Java does allow the method in the derived class to have a *less restrictive* access control than the method it overrides from the base class. However, a `private` method in the base class cannot be overridden, since it is not visible to any code except the class in which it is defined. A method with the same name and signature in the derived class would simply be a method of that class, and would not override the `private` method in its superclass.

Try it out:

`https://github.com/SienaCSISAdvancedProgramming/OverrideAccessControl`

Covariant Return Types

Normally, we would expect the return type of a method in a subclass that overrides a corresponding method in a superclass to be the same as the return type in the superclass. We are allowed to modify the return type of the overriding method, as long as it is a subtype of the original return type. This is called a *covariant return type*.

This can help reduce the amount of type checking (with `instanceof`) and casting in some circumstances.

See an example at `http://javapapers.com/core-java/covariant-return-type-in-java/`

`final` and Inheritance

The `final` keyword can also be used to place restrictions on inheritance and function overriding.

If the `final` qualifier in a class header, it means that the class cannot be extended:

```
final class A { }

class B extends A { }
```

This results in a compile error.

`final` can also be used on an individual method to prevent that method from being overridden in any subclass:

```
class A {

    public final int x() {

        return 1;
    }
}

class B extends A {

    public int x() {

        return 2;
    }
}
```

This also results in an error at compile time.

Declaring a class as `final` is useful in cases where objects of the class are intended to be *immutable*, like `java.lang.String`.

Polymorphic Names

We have seen the idea of *polymorphism*, the same name referring to different entities. Now, let's look at five terms related to polymorphism and how they related to Java.

- overloading
- overriding
- shadowing
- hiding
- obscuring

Examples in this section can be found in

`https://github.com/SienaCSISAdvancedProgramming/PolymorphicProblems`

We discussed *overloading*, or *ad hoc polymorphism*, earlier. This is when two methods of the same class have the same name but different method signatures. Java can decide which to use based on the method signature when it is called.

We soon after discussed *overriding*, where a method in a derived class replaces the version in would otherwise inherit from a base class. We also looked carefully at how Java determines which method gets called in various situations, given that it supports *dynamic binding*.

We have not necessarily used the term *shadowing*, but you have certainly experienced it. This occurs when a parameter or local variable has the same name as an instance variable, as shown in the example below.

```
public class A {
    private int x;

    public A(int x) {
        this.x = x;
    }

    public void f(int y) {
        int x = y*2;

        this.x = y;
    }
}
```

In the constructor, we see the familiar pattern where we access the instance variable `x` that is shadowed by the parameter `x` by referring to it ad `this.x`. In the method `f`, we see the same shadowing problem with the local variable `x`, but we can use the same technique to access the instance variable as `this.x`.

If we have a subclass with the same variable name as its superclass, we have an example of *hiding*. In `Hiding.java`, we see that both class `A` and class B (which extends `A`) have an instance variable named `x`. `B` actually has access to both in this case, as it can use `super.x` to get around this case of hiding, much like we use `this.x` to get around shadowing.

This can occur by mistake when variables are factored out with code into base classes or abstract classes and we forget to remove them from the derived class(es).

Finally we get to *obscuring*, which can result from a class and a variable having the same name. Consider this example from

`https://programming.guide/java/overloading-overriding-shadowing-hiding-obscuring.html`

```
class C {
    void m() {
        String System = "";
        System.out.println("Hello World");
    }
}
```

This would not compile! The local variable `System` obscures the class name `System`. We could get around it by replacing using the class `System`'s fully qualified name: `java.lang.System`.

The `PolymorphicProblems` repository has two of Dr. Lim's programs that we will check out that are examples of obscuring, in `Obscuring.java` and `Obscuring2.java`.

If you run the former, you will notice that `System.out.println` in `main` does *not* result in a call to the standard `System.out.println`!

A second example shows obscuring of the constant `MIN_PRIORITY` from the `Thread` class:

Note that in both of these cases, we can still access the obscured name by using its fully qualified form.

Initializer Blocks

Have you ever intentionally or unintentionally included some Java code outside of any method? You might have forgotten your `public static void main` method header and it's possible your code actually compiled!

Our final example will explore the ideas of *initializer blocks*, which can be `static` (which would be a *static initializer block*) or not (which would be an *instance initializer block*), and see when those get executed relative to other methods and constructors.

`https://github.com/SienaCSISAdvancedProgramming/Initializers`

`Initializers.java` shows code with two non-static initializer blocks defined, and has a `main` method we can run to see when that code gets executed.

`StaticInitializers.java` adds a static initializer block and has overloaded constructors. Again, we can run its `main` method to see when each of those parts of the code gets executed.

`InheritanceInitializers.java` introduces inheritance to the situation. Again, we can run its `main` method to see when each of those parts of the code gets executed.