Visibility and Constructors

(These are especially necessary for OOP.)

OOP Review

Inside our classes, we can create variables and methods to store the data and the procedures for each instance of the class.

We can access variables within an instance using the dot operator, like this: `song1.rating = 5;`

Methods have return types (including void) that specify the type of data that will be returned to the caller. Methods can also have arguments that specify the required inputs when they are called.

For example:

```
public boolean isOdd(int number)
```

Privacy Issue

We may have variables in a class that we do not want modified. For example, look at this code that cheats at a card game:

```
PokerHand myCards = dealer.dealCards();
myCards.card[1] = new Ace();
```

It would be nice if we didn't have to worry about someone modifying the card array inside a PokerHand instance.

In general, we need a way to allow programmers to use objects from a class without worrying that they will accidentally (or intentionally) change the object's variables in a way that the original programmer did not intend.

Visibility Keywords

Java provides access level keywords: `public` and `private`

You can mark classes, methods, and instance variables with these keywords:

```
public class Vault
{
    private String secret;
    public void keepSecret(String s)
    {
        secret = s;
    }
}
```

Public

When you mark something public, it means that it can be used from any context. So public variables can be read and written from outside the class:

```
Dog fluffy = new Dog();
// the next two lines are OK if name is public
fluffy.name = "Bob";
System.out.println(fluffy.name);
```

Public methods can be called from outside the class:

```
// the next line is OK if bark() is public
fluffy.bark();
```

Private

When you mark something private, it means it can only be used within the class. Private variables can only be read or written by code in methods in the same class. Private methods can only be called from other methods in the same class.

```
Dog fluffy = new Dog();
// neither of the following two lines will
// compile if name is private
fluffy.name = "Bob";
System.out.println(fluffy.name);

// and this call fails if bark()is private
fluffy.bark();
```

Getters

Usually, instance variables are marked private. However, we also often want to read the value of the variable from outside the class. If you decide to allow this, use the Getter method pattern:

```
public class Cat
{
    private int age;

    public int getAge()
    {
        return age; // yep, it's this easy
    }
}
```

Setters

Like Getter methods, Setter methods are used with private instance variables. In a Setter method, you should test and validate the argument value to enforce appropriate restrictions before assigning to the private variable.

```
public class Cat
{
    private int age;
    public void setAge(int a)
    {
        if( a>0 ) age = a;
    }
}
```

Encapsulation

Wikipedia says:

"Encapsulation is the packing of data and functions into a single component. The features of encapsulation are supported using classes in most object-oriented programming languages, although other alternatives also exist."

"Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. A supposed benefit of encapsulation is that it can reduce system complexity, and thus increase robustness, by allowing the developer to limit the inter-dependencies between software components."

Basically, encapsulation encourages information hiding. Always make your instance variables private to prevent bugs and abuse.

Class Methods

Given a class definition, we can make many instances:

```
Dog rover = new Dog();
Dog fluffy = new Dog();
```

And then we can set the properties of each instance:

```
rover.setName("Roverius");
rover.setWeight(25);

fluffy.setName("Killer");
fluffy.setWeight(4);
```

Problems with new objects

1. Creating objects and then setting their properties is annoying: we have to type multiple lines just to make a working object.

2. EVEN WORSE, it's possible to create an object that does not have its important variables set:

```
Dog x = new Dog();
System.out.println( x.getName() );
System.out.println( x.getWeight() );

// outputs a blank line
// and then a 0
```

Let's combine them

Instead of these three lines:

```
Dog fluffy = new Dog();
fluffy.setName("Killer");
fluffy.setWeight(4);
```

Wouldn't it be nicer to just type this?

```
Dog fluffy = new Dog("Killer", 4);
```

Can we do that?

Constructor methods

To facilitate this new syntax, all we have to do is provide a *constructor*. For example:

```
public class Dog extends Pet
{
  public Dog( String n, int x )
  {
    setName(n);
    setWeight(x);
  }
  // methods, etc. not shown
}
```

Constructors are special

Constructors are like methods in some ways:

- They require a parameter list with 0 or more parameters
- They can be overloaded, so one class can have many
- They can be set public or private

Constructors are different than methods in some ways:

- They do not have a return type
- They can't be called directly by name
- Instead, we use the **`new`** keyword
- Later we'll also learn how to call via **`this()`** and **`super()`**

When does a constructor run?

Like any object declaration/creation/assignment, this statement has three parts:

```
Dog fluffy = new Dog("Killer", 4);
```

1. Variable declaration
2. Object creation (when the constructor runs)
3. Assignment of the new object reference to the variable

How would it work without a constructor?

So back when we were writing stuff like:

```
Dog fluffy = new Dog();
fluffy.setName("Killer");
fluffy.setWeight(4);
```

What did the JVM do when it got to the `new Dog()` part?

The default constructor

If you don't provide a constructor, the compiler will add one for you, and it will look like this:

```
public class Dog extends Pet
{
  public Dog()
  {
    super();
  }
  // methods, etc. not shown
}
```

So we have unknowingly been using constructors all along.

We need super()

The call to **`super()`** causes the superclass constructor to run.

But didn't we just write another constructor without that?

```
public Dog( String n, int x )
{
  setName(n);
  setWeight(x);
}
```

We did, and again the compiler adds something without us knowing about it...

super() default

The compiler automatically adds a call to **`super()`** at the top of any constructor that doesn't have one:

```
public Dog( String n, int x )
{
  super();
  setName(n);
  setWeight(x);
}
```

Super confusing? (pun intended)

So the compiler is doing a lot of stuff for you. The good news is that it's usually what you want. The bad news is that it's sometimes not.

For now, let's take advantage of the power of constructors to become familiar with how they work.

Any time you find yourself calling additional methods after creating an object in order to set its initial state, use a constructor instead.