CSCI 112
Wednesday, July 7, 2010
Class Notes

Review of Last Class

Vocabulary

Class – plan/blueprint which defines an object
Object – a collection of variables and associated methods
Method – an action performed on an object
Constructor – a method used to instantiate an object
Parameter – input to a method, sometimes called arguments
Return Type – data type of the method
Private Field – field that cannot be seen outside of the class
Public Field – a field that can be seen by any class
Static Field – associated with the class
Non-Static Field – associated with the object

Subclasses

- Primary example of code reuse.
- **DRY**: don't repeat yourself, keep your code "DRY."
- In a subclass, we are taking a class and making it more specific.

Subclass – derived class, child class
- the new class

Base class – super class, parent class
- old class being extended

IS A – relationship between child and parent classes.

The only way we can derive a class is if the following sentence makes sense:
"Child Class is a Parent Class."
If so, you can successfully make a child class.

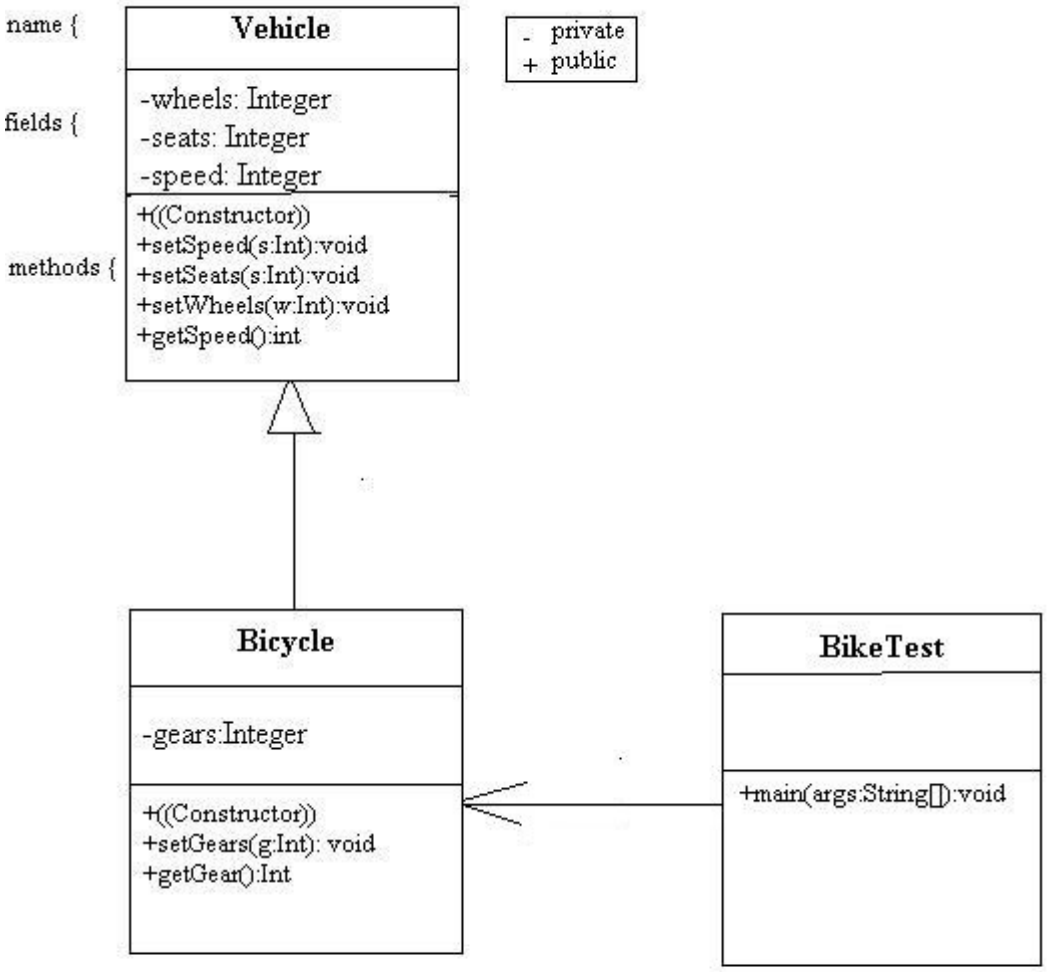
Examples:

Correct: "Horse is a mammal"
Incorrect: "Country is a USA"

Unified Markup Language ("UML")

Two Symbols to Remember in UML diagrams:
- A triangle at the end of a line means **extended class.**
- A line with an arrow means **accesses this class.**

see 100707 Table1.jpg *for example*



100707 Table1.jpg

Public, Private, Protected, Package

**This list is ordered from most restrictive permission to least restrictive permission*

- **Private:** means that only methods within this class can access this method or field.
- **Protected:** means that methods found in this class or classes that extend this class may access this field.
- **Package:** (the default, has no keyword) means that any method in any class found in this directory may access this method or field.
- **Public:** means that any method in any other class can access this method or field.

Example Code (Vehicle.java)

```
public class Vehicle{
	private int wheels;
	private int seats;
	private int speed;

	public Vehicle(){
		this.wheels = 0;
		this.seats = 0;
		this.speed = 0;
	}

	public void setSpeed( int speed ) {
		this.speed = speed;
	}

	public void setSeats( int seats ) {
		this.seats = seats;
	}

	public void setWheels( int Wheels ) {
		this.Wheels = Wheels;
	}

	public int getSpeed(){
		return speed;
	}
}
```

Bicycle.java

```
public class Bicycle extends Vehicle{
```

```
        private int gears;

        public Bicycle( int gears ){
                setWheels( 2 );
                setSeats( 1 );
                this.gears = gears;
        }

        public void setGears( int gears ){
                this.gears = gears;
        }
        public int getGears(){
                return gears;
        }
}
```

BikeTest.java

```
public class Biketest{
        public static void main( String[] args ){
                Bicycle b = new Bicycle( 1);
                b.setGears( 2 );
                b.setSpeed( 10 );

                System.out.println( "Bike Speed: " + b.getSpeed );
        }
}
```

Example of An Extended Class, More Features

Book.java

```
public class Book{
        protected int pages = 1500;
```

```
	//Default constructor, if you don't have one, Java will create one for you
	public Book(){
	}

	//Initialization Contructor.  Initializes the fields.
	public Book( int pages ){
		if( pages >= 1 ){
			this.pages = pages;
		}
	}

	public void setPages( int pages ){
		if( page >= 1 ){
			this.pages = pages;
		}
	}

	public int getPages(){
		return pages;
	}
}
```

Dictionary.java

```
public class Dictionary extends Book{
	private int definitions = 52500;

	//Default Constructor
	public Dictionary(){
		super(); //calls default base constructor
	}

	//Initialization Construtor
	public Dictionary( int pages, int definitions ){
```

```
                super(pages); //call initialization constructor

                if( definitions >= 1 ){
                        this.definitions = definitions;
                }
        }

        public void setDefinitions( int definitions ){
                if( definitions >= 1 ){
                        this.definitions = definitions;
                }
        }

        public int getDefinitions(){
                return definitions;
        }

        public double computeRatio(){
                return (double)definitions/pages;
        }
}
```

Words.java

```
public class Works{
        public static void main( String[] args ){

                //first dictionary
                Dictionary basic = new Dictionary();

                System.out.println( "Basic, pages: " + basic.getPages () );
                System.out.println( "Basic, words: " + basic.getDefinitions() );
                System.out.println( "Basic, WPP: " + basic.computeRatio() );
```

```
        //second dictionary
        Dictionary oxeng = new Dictionary(2000, 80000);


        System.out.println( "Oxford, pages: " + Oxford.getPages () );
        System.out.println( "Oxford, words: " + Oxford.getDefinitions() );
        System.out.println( "Oxford, WPP: " + Oxford.computeRatio() );
    }
}
```

Inheritance in Other Languages

- **Java** is a *single inheritance* language. This means a class can only derive from one other class.
- **C++** is a *multiple inheritance* language. This means a class can derive from multiple classes. This leads to several problems. One main problem in the C++ language is known as the **diamond problem**.

Diamond Problem

Say that class A and B extend from D and class C extends from A and B. If you want to call a method from D from within C, your program doesn't know how to get there (through A or B), so the program crashes.

So Where is Multiple Inheritance Useful?

Say you have a gas engine and an electric engine. A Hybrid engine inherits from both. There is no way to express this in the Java language. So in Java you can not make efficient reuse of code because you can't inherit from both at the same time, requiring you to code all the code from both into Hybrid.

Traits

A third type of inheritance. Ruby is able to make use of this type. Instead of creating a class, you create traits. Then you allow say your hybrid engine to have traits of both gas and electric. A trait can be seen as a sort of crippled class.

This was introduced in Ruby using the relationship "**like a.**"

Java is not able to make use of this either.