Inheritance

Introduction

This unit covers to how to make classes inherit properties from other classes and files.

Subclasses

When building classes many times you need a new class that is modification of an existing class or an extension of it.

The class that will be extended from is called the **parent class** or **superclass**. The new class that is being created from the parent class is called the **child class** or **subclass**.

The parent's parent or its parents are known as **ancestors** of the child class. The child's children or their children are known as **descendants** of the parent.

When you **extend** a class the child class will be given all of the parent's attributes, methods, and access to the parent's constructors. The act of gaining everything thing from the parent class is called **inheritance**.

What you need to know about inheritance:

- A class can only have one parent class
- Your gain all the attributes & methods of your parent class and any ancestors
  - Although you can only access the public ones.
- The first line of in the constructors must be a called to the parent's constructor
- If you want to change a parent's or ancestor's method it can be done through a process known as **method overriding**.
  - **Note:** when you override a method you should place **@override** above the method. This will have the compiler create an error if you mess up the method header.
- Every class is derived from the Object class
  - Methods Gain from Object
    - equals(Object) – true when the calling object and the received object have the same memory address, false otherwise.
      - Should be rewritten to compare the data of the two objects
    - toString() – Returns the names the class name of the object and it's memory address
      - Should be rewritten to return important pieces of the class's data

Format for Extending a Class:

```
public class ClassName extends ParentClassName
{
}
```

Format for Calling the Parent Class Constructor:

```
super(parameters);
```

Format for Calling a Method of a Parent Class:

```
methodName(parameters);
```

Format for Calling a Method of a Parent Class (when the method has been overwritten):

```
super.methodName(parameters);
```

Example:

```
public class Person
{
        private String firstName;
        private String lastName;

        public Person(String firstName, String lastName)
        {
                this.firstName = firstName;
                this.lastName = lastName;
        }

        public String getFirstName()
        {       return firstName;       }

        public String getLastName()
        {       return lastName;        }

        public String toString()
        {
                String s = “”;
                s+= “First Name:\t”+ firstName + “\n”;
                s+= “Last Name:\t”+ lastName + “\n”;
                return s;
        }
}

public class Student extends Person
{
        int student id;

        public Student(String firstName, String lastName, int id);
        {
                super(firstName,lastName);
                this.id = id;
        }

        public int getID()
        {       return id;      }

        public String toString()
        {
                String s = “”;
                s+= super.toString();
                s+= “ID:\t\t”+ id + “\n”;
                return s;
        }

}
```

Interfaces

Interfaces are basic files that work as a requirement list for other classes. Interfaces can only contain public static final attributes and **abstract methods**. Abstract methods are methods that contain only the method header followed by a semicolon. These methods must be completed by any class that implements the interface.

Format for Writing an Interface:

```
public interface FileName
{
}
```

Format for an Abstract Method:

public abstract <u>type</u> <u>methodName</u>(<u>parameters</u>);

Notes:

- Abstract Methods cannot be static or final
- The keyboard abstract is not required when writing an abstract method, but it recommended to add it.

Format for Implementing an Interface:

```
public class ClassName implments InterfaceName
{
}
```

Notes:

- A class can implement an unlimited number of interfaces.
- To implement more interfaces list off all the interfaces separating them with a comma.

Example:

```
public interface Clearable
{
        public abstract void clear();
}

public CheckOut implements Clearable
{
        private ArrayList<Items> items = null;
        private double subtotal;
        private double shipping;
        private double tax;
        private double total;

        public CheckOut(ArrayList<Items> items)
        {
        …
        }
        …

        public clear()
        {
                items = null;
                subtotal = total = tax = shipping = 0;
        }
}
```

Abstract Classes

Abstract classes are exactly like classes with two main differences. Abstract classes can contain abstract methods and instances of abstract classes cannot be created. Any class that extends an abstract class will gain all its methods attributes, constructors, methods and must write each abstract method the abstract class contains.

Format for Writing an Abstract Class:

```
public abstract class FileName
{
}
```

Format for Extending an Abstract Class:

```
public class ClassName extends AbstractClassName
{
}
```

Storing Objects

A variable can store any class that is of its type or any class that is derived from it.

Format for Storing an Object:

```
StaticType name = new DynamicType();
```

Examples:

```
Object name = new String("bob");     // All Objects come from Object
Comparable number = new Integer(5); // Integer implements Comparable
```

The **static type** is the type of the variable and is the type used when the program is compiling.

Name has a static type of Object so name.toString() would compile, but name.charAt(2) would not. This is because the static type is used to determine what methods can be called.

Name is a String so we can force the compiler to allow charAt to be called by casting name to a String.

Fixed charAt Call:

```
((String)name).charAt(2);
```

The **dynamic type** is the type of the object that was actually created. For name the dynamic type is String. When the program is running the dynamic type is used.

For example, if a Student were stored into an object variable its static type is Object and its dynamic type would be Student. If we were to call the toString() method on our object it would compile since Object has a toString() method. When the program is run toString() would return the first name, last name and id of the student, due to its dynamic type.

instanceof

Sometimes while your program is running you will need to check the class of data in variable. To do this we will use the instanceof operator. instanceof returns a boolean value letting the program know if a variable is of a given type or not.

Format for using instanceof:

<u>varaibleName</u> instanceof <u>type</u>

Example:

```
if(data instanceof Integer)
        System.out.println("It is a number");
else if(data insanceof String)
        System.out.println("It is a String");
else
        System.out.println("It is not a String or number.");
```

Differences in File Types

Things allowed	**Class**	**Interface**	**Abstract Class**
public static final attributes	X	X	X
static attributes	X		X
non-static attributes	X		X
constructors	X		X
static methods	X		X
non-static	X		X
abstract methods		X	X
Instances of it can be created	X		

Terms

Abstract Class	A class that has one or more abstract methods.
Abstract Methods	Methods that have only a method header and will be written in another file.
Ancestor	Any class above a class’s parent.
Child Class / Subclass	The class doing the extending.
Descendant	Any class below a class’s child.
Dynamic Type	The runtime type for an object.
Extending	Setting a class’s parent class.
Inheritance	The gaining attributes, constructors and methods from a class’s parent and ancestors.
instanceof	A keyword used to determine if a varaible’s data is of a given type.
Interface	A file that stores methods a class must write to fit a given category.
Method Overriding	Overriding a parent’s or ancestor’s method by rewriting it.
Parent Class / Superclass	The class that is being extended from.
Static Type	The compile time type for a variable.

Labs

(0) Level A	(1) Level B	(1) Level C
	Connect Four Players	Video Store