Introduction to Inheritance - (JAVA)

Inheritance is a language construct that supports the sharing of features amongst different objects. Consider the domain of vehicles, which includes bicycles, skateboards, cars and jets. On the one hand, vehicles of these types share some common features; they tend to be manufactured by particular companies and identified by a model name or number. For example, the "7.4FX" is a particular bicyclemodel manufactured by Trek Corporation and the "Rocket" is a particular skateboard model manufactured by Ally Corporation. On the other hand, each of these vehicle types tends to have distinguishing features not shared by other vehicle types. For example, bicycles can be assessed by their number of gears, e.g., 27 for the Trek 7.4FX, while skateboards can be assessed by the length of their board, e.g., the Rocket has a 31.5-inch board.

Inheritance allows a programmer to separate those attributes and behaviors that are shared between vehicle types and those that are unique to each particular type. The shared features are collected in a single class known as the parent or superclass and the unique features are separated into the child orsubclasses. This can be visualized as follows.

In class diagrams such as this, subclasses point up to their superclass. The attributes and behaviors implemented in the superclass are "inherited" by all the subclasses. The attributes and behaviors implemented in one of the subclasses are unique that subclass. In a sense, the features shared by subclass1 and subclass 2, that might otherwise have been implemented separately in each of the subclasses, can be collected and "raised up" into the single shared superclass.

Because Java does not implement multiple inheritance, subclasses can only have one superclass. Superclasses, on the other hand, can have many subclasses.

For example, in the vehicles domain, a programmer might implement the brand and model in a vehicle superclass, the engine size in a car subclass and the number of jet engines in a jet subclass.

The attributes and behaviors implemented in the superclass are "inherited" by all the subclasses. The attributes and behaviors implemented in one of the subclasses are unique that subclass. In a sense, the features shared by subclass1 and subclass 2, that might otherwise have been implemented separately in each of the subclasses, can be collected and "raised up" into the single shared superclass.

Because Java does not implement multiple inheritance, subclasses can only have one superclass. Superclasses, on the other hand, can have many subclasses.

The extends Clause

Inheritance is implemented in Java using the `extends` clause. A class `Subclass1` can inherit attributes and behaviors from another class `Superclass` as follows:

```
class Superclass {
// attributes and behaviors shared by all subclasses...
}
class Subclass1 extends Superclass {
// attributes and behaviors unique to Subclass1...
}
```

The `extends Superclass` clause specifies the inheritance. It indicates that any object of type`Subclass1` is also an object of type `Superclass` and thus that a `Subclass1` object can do anything that a `Superclass` object can do. This construct provides considerable power for code sharing and reuse.

The Java Class Hierarchy

We have actually been taking advantage of inheritance all along because Java structures its entire API hierarchically[1]. There are far too many classes – more than 4000 in Java 7 – to show, but the root of this hierarchy is the `Object` class, making it the common ancestor for all Java classes

Note that `PApplet` as well as all the other Processing-specific classes are integrated into the Java class hierarchy.[2] In addition, every new class that any programmer writes is made an extension of the `Object`class by default, regardless of whether the programmer explicitly includes the `extends Object` clause or not.

Because of this every class inherits the features of the `Object` class, which include the `toString(),clone()` and `equals`() methods. This explains why we can always print an object on the console; every class inherits the `toString()` method from the `Object` class automatically.

Accessing Superclass Constructors

The Java class-construction mechanism provides two useful keywords: `this` and `super`. Both are discussed in this section with respect to their use in accessing constructor methods.

The `this` Keyword

The keyword `this` refers to the current object itself. It is occasionally used to access one constructor from another and also to access data attributes that are out of scope, as shown here.

```
class A {

private int someValue;

public A() {this(1);

}

// Call the explicit-value constructor.

public A(int someValue) {
this.someValue = someValue;// Access the class instance variable.

}
}
```

In this code, the default constructor uses `this` to access the explicit-value constructor, passing a default value for the instance variable. The explicit-value constructor must use `this` to access the class instance

variable because the parameter of the same name overrides the global definition in the scope of the explicit-value constructor.

The `super` Keyword

The keyword `super` refers to the superclass of the current object. It performs a similar function, but it refers to the immediate superclass of the current class. This is useful when a class needs to access: (1) itssuperclass's constructor, discussed in this section; and (2) its superclass's methods, discussed in the nextsection.

While subclasses inherit the attributes and methods of their superclass, they do not inherit theirsuperclass's constructors. To invoke a superclass's constructor, a subclass must use the `super` keyword as shown here.

```
super(argumentList);
```

The argument list provides the arguments required by the superclass's constructor and may be empty forthe default constructor. Note that this call to the superclass's constructor must be the first statement in a subclass's constructor.

For the vehicle domain, a programmer will likely want to improve the current implementation of the`Bicycle` class by providing an explicit-value constructor that specifies all the instance variables. This constructor would be invoked as follows.

```
Bicycle feltAR5 = new Bicycle("Felt", "AR5", 30);
```

The current version of the `Bicycle` class does not provide a constructor with this signature because it cannot access the `myBrand` and `myModel` variables it inherits from the `Vehicle` class. They are declared as `private` data items and cannot, therefore, be accessed by any class other than `Vehicle`, even `Vehicle`'s own subclasses.3

Instead, a programmer can use the `super` keyword to access the features of the superclass's constructor method. Given the definition of the `Vehicle` class shown in Section 13.3.1, which includes an explicit-value constructor that receives two strings representing the brand and model respectively, we can revise the `Bicycle` class as shown here.

Overriding Superclass Methods

Subclasses are not required to use the same definitions of all the methods they inherit; they can replace them with specialized definitions. This is called overriding the inherited method

A programmer can override an inherited method as follows.

```
class Superclass {

            // other features of the superclass...      public int superClassMethod() {
// do something...

return 1; }

}
class Subclass extends Superclass {

            // other features of the subclass...      public int superClassMethod() {
// do something different...          return 2;

}
```

In this code, the subclass provides its own definition of the superclass's `superClassMethod()`. An object of type `Subclass` will execute its own version of `superClassMethod()`, that is it will "do something different" and return 2 rather than 1.

Note the difference between "overriding" a method as described here and "overloading" a method as described in a previous chapter. In contrast to overriding, overloading a method means that we've defineda new method with the same name but a different signature. Constructor methods are commonly overloaded.