Inheritance and Polymorphism Chapter

Contents

[The UML symbols: 1](#_Toc52691546)

[Inheritance 1](#_Toc52691547)

[extends 3](#_Toc52691548)

[super 3](#_Toc52691549)

[Java’s Inheritance Chain 3](#_Toc52691550)

[Overloading vs Override 4](#_Toc52691551)

[toString 5](#_Toc52691552)

[equals 5](#_Toc52691553)

[Polymorphism 5](#_Toc52691554)

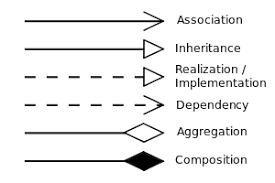
[Dynamic Binding and Static Binding 5](#_Toc52691555)

[instanceof 6](#_Toc52691556)

[Visibility Modifiers 6](#_Toc52691557)

[final 7](#_Toc52691558)

# The UML symbols:



# Inheritance

Inheritance is describing a class relationship between two classes.

Inheritance means to define a new class from existing classes.

It is one of the pillars of OOT

* Utilizes the keyword extends
* Means you can reuse existing code
* The class that is being inherited **from** is the superclass
* The class that is inheriting (obtaining all the states and behaviors) is the sub class
* Therefore the subclass is a specialized class containing the superclass’ states and behaviors **and** also its own states and behaviors
* Java is single inheritance
* Java uses inheritance chain

Inheritance happens between two classes, when one class *is-a* of the other class; forms an *is-a* relationship.

AKA:

Superclass = parent class = base class

Subclass = child class = derived class = extended class

Why do you want to use inheritance?

Because it allows you to create a common class that can be used by several other classes -

* meaning, you only have to define these states and behaviors only once & get to use them the other classes
* benefits: reuse software, this makes the software modularized which inturn makes it easier to maintain AND easier for us programmers to comprehend/understand the code

When instantiating a subclass object, the JVM must first invoke the superclass’ constructor, then it can create the subclass object.

This is called constructor chaining.

***If no inheritance is specified when a class is defined, then it inherits from Object by default.***

***All*** objects (behind the scene) inherit from the Object class along the inheritance chain.

When an object inherits from another object, that inheriting object becomes more specific of the inherited-from object, meaning you are adding states and behaviors to the inheriting object.

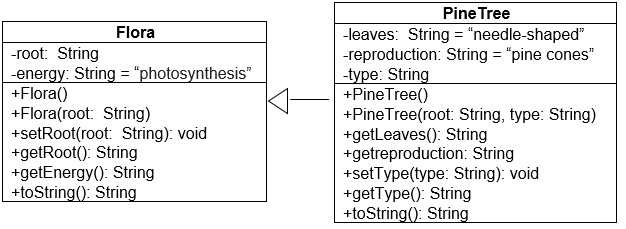
It is important to know that a subclass *is not a subset* of a superclass.

A subclass has all of the states and behaviors of a superclass and then it has its own states and behaviors.

Not all relationships are an *is-a* relationship, be diligent in understanding the relationships between objects.

An example: a Flora class and a PineTree class

A PineTree *is-a* Flora, a specialized class of Flora



## extends

Thus PineTree class code uses extends

public class PineTree extends Flora{

Then the compiler and the JVM knows that root, energy, and setRoot(), getRoot(), getEnergy(), and toString are inherited and PineTree has those states and behaviors as well as leaves, reproduction, getLeaves(), getReproduction(), setType(), getType(), and toString().

When the JVM executes the statement

PineTree pt = new PineTree();

It first goes to the PineTree class, sees that it extends the Flora class; this invokes the Flora’s constructor. This means a Flora object ***has to be*** created first. (Behind the scene, the Flora’s constructor invokes Object’s constructor which is in reality created first, then the Flora, the the PineTree.) Then the JVM returns to the PineTree constructor to instantiate (“add/build” onto) a PineTree object.

This leaves the question of how to access the superclass’ states and behaviors. The states use the visibility modifier of private and cannot be directly access (remember encapsulation!), but the keyword super can access the super class.

## super

The keyword super allows subclasses to invoke the superclasses’ methods, e.g. the accessors and mutators are public (the proper channel, thus the object’s data is protected).

super also allows to pass the states of the superclass onto to the superclass.

Examples:

PineTree’s constructor:

public PineTree(String root, String type){

super(root);

this.type = type;

}

super(root) then passes the variable root to the superclass, Flora’s constructor.

Accessing the methods:

public String toString(){

return super.toString() + this.type + “ kind ” + “ and has “ + this.leaves + “ leaves, and “ + this.reproduction;

}

Java’s Inheritance Chain:

Java has single inheritance, but you can extend a class which extends a class… this is the inheritance chain. And gives you constructor chaining.

When an object extends (inherits from) another object, then the superclass has to be created first, prior to creating the subclass

White

For example, it you have

Grey

Purple

Then, to create a Purple object, the JVM must invoke the Grey object’s constructor; but before it can create the Grey object, it must invoke the White object’s constructor; but before it can do that, it invokes Object’s constructor (creating/instantiating an Object), then it returns to White’s constructor (creating/instantiating-adding to Object, a White object), then returns to Grey’s constructor (creating/instantiating– adding to a Grey object), then returns to Purple’s constructor (creating/instantiating– adding to a Purple object).

This means that:

a Purple object is an instanceof a Grey object and a White object and an Object object

a Grey object is an instanceof a White object and an Object object

# Overloading vs Override



Both use the method signature, the diffence is how the method signature is defined.

**Overloading**: uses the same method name and has *different* formal parameters; this is when the methods are in the same class/program

**Override**: uses the same method name and the *same* formal parameters; this is when the classes are in an inheritance chain

* use @Override **prior** to the method header: this lets programmers know that the method has been overridden and assists the JVM during runtime and polymorphism
* **NOTE**: you ***cannot*** override a static method
  + Remember what the modifier static represents – available to all - overriding a static defeats the purpose of being available to all

## toString

The Object class has several methods that all ADT have by default then, the toString is one.

The toString method is to provide a description of the ADT, it returns a String that describes the object.

The toString method in the Object class returns the object’s memory address, which is not descriptive; hence, ***you have to override this method***.

@Override

public String toString(){

return “the description of the object/ADT.” ;

}

FYI: System.out.println(object); and System.out.println(object.toString()); are the same / equivalent.

## equals

Object’s equals method is another method that all ADTs have.

The default equals method uses the primitive comparison of ==

This means that it is looking at the contents that are sitting in main memory, which works great for primitive datatypes but not object datatypes.

You should override this method, so that your objects are being evaluated correctly. (see TestFloraPineTree.java for examples)

# Polymorphism

One of the pillars of OOP

If we look at parts of the word:

Poly = many

Morph = change

And if we take that and apply that to how we are learning about objects, roughly translates to an object that can change.

Polymorphism means that a variable of a supertype can refer to a subtype object; the subtypes have the states and behaviors of its supertypes, but supertypes do NOT have the states and behaviors of any subtypes

Since all subtypes have the state and behaviors of it supertypes; the sypertypes can refer to a subtype. It CANNOT be the other way around.

So a valid statement is: Grey g = new Purple();

**NOT** valid: ~~Purple p = new Grey();~~

Why do we want to use polymorphism? Polymorphism helps simplify the code and provides flexibility; you can create one method to that can access all the states of the superclass.

## Dynamic Binding and Static Binding

**Dynamic Binding**: the JVM deciding which method to invoke during runtime

**Static Binding**: the compiler is matching call statements with method headers

Polymorphism happens during runtime, meaning that dynamic binding is used.

Static binding is during compile time, the compiler of matching method names with the formal parameter list.

Dynamic binding happens during runtime, the JVM is figuring out which class’s methods to use based on the super or sub types.

Since an object can refer to its subtype, then the issue lies in how to access the subtypes states and behaviors.

## instanceof

You as a programmer want to avoid runtime errors! Using the instanceof in a Boolean test in if statements ensures the object is that subtype, then you cast it to the subtype (so the JVM will go to the correct object) to access the states or behaviors.

Example:

if (g instanceof Purple){

System.out.println( ((Purple)g).getPrefix() ); //added spaces to see the casting

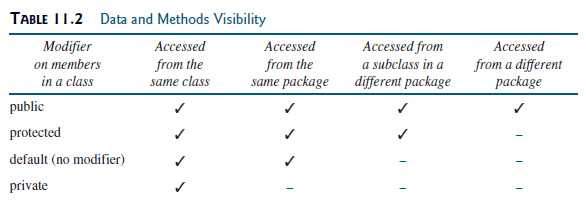
}

Looking the example:

It is not an issue to access the supertypes states and behaviors.

System.out.println( g.getNum() );

# Visibility Modifiers



UML symbols:

public +

protected #

default (none)

private –

The protected visibility modifier allows access of states and behaviors in a superclass from its subclass.

Keep in mind encapsulation! Using the private modifier creates encapsulation for that class, protected allows a modified encapsulation by allowing the subclasses access to the superclass data.

# final

The final keyword prevents changes.

Just like in your learned in CSCI 1302, final prevented a variable from having the value changed, final can prevent changes to a class and to a method.

Examples:

private int NUM\_STAYS\_THE\_SAME = 42; // the variable cannot be changed

public final void cannotOverrideMethod()// the method cannot be overridden

public final ClassCannotBeExtended // the class cannot be a superclass, if it is a

subclass it is the end of the inheritance chain