Java Chapter 10 - Intro to Inheritance

**INHERITANCE:**

* Enables one class to acquire all behaviors and attributes from another class
* Classes created can inherit data and methods from existing classes
* The new class automatically contains the data fields and methods of the original class
* Diagramming Inheritance:
* Programmers & analysts use a graphical language to describe classes & object-oriented processes 🡪 UML 🡪 UML Class Diagram
* Usage:
* Makes a classes code more easily reusable
* Makes programs easier to write, less error prone, & more easily understood
* Distinguish Between:

\*\*Base/Super/Parent Class & Derived/Sub/Child Class:

* Derived class always “is a” case / example of the more general base class

Ex 🡪 Evergreen is a Tree

* Don’t confuse “is a” situations with “has a” situations (composition)

Ex 🡪 a Business has Departments

* Say the Class names together 🡪 Evergreen Tree
* By size 🡪 subclass generally larger than superclass
* Extending Classes:
* Use the keyword extends to achieve inheritance in Java

Public class EmployeeWithTerritory extends Employee

* Statement that instantiates a derived class object:

EmployeeWithTerritory northernRep = new EmployeeWithTerritory();

\*This Object now has access to all methods in the Employee Class

* **instanceof() :** operator used to determine whether an object is a member or descendant of a class

northernRep instanceof EmployeeWithTerritory 🡪 true

northernRep instanceof Employee 🡪 also true

\*Programmers say that instanceof yields true if the operand on the left can be upcast to the operand on the right \*

**OVERRIDING SUPERCLASS METHODS:**

* When the superclass data fields & methods are not entirely appropriate for the subclass objects
* OVERRIDE a field or method 🡪 using the child’s version instead of the parent’s version
* When you create a method in a child class that has the same and parameter list as a method in its parent class aka override the parent
* When you use the method name with a child object, the child’s version is used
* Polymorphism:
* Any operation that has multiple meanings (**‘+’** 🡪 used in both addition & concatenation,

methods w/ same name & different parameter lists – **overloading**)

* Subclasses Guitar and Drum 🡪 carry out a play() method differently
* Superclass: Employee
* Subclasses: HourlyEmployee & ContractEmployee

These each require their own paycheck-calculating procedures

\*Each subclass method **overrides** any method in the parent class that has both the same name and parameter

\*If the parent class method has the same name but a different parameter list, the subclass methods **overloads** the parent (not override)

\*If you could not/didn’t want to override a superclass method 🡪 could always just create a unique name for each subclass method

DisplayRateOfPayForHourly() & displayRateOfPayForContract()

* @Override Tag:
* Can insert an optional override annotation prior to method header
* Announces your intention to override a parent class method and causes the compiler to issue an error message if you do not
* Prevents errors & Documents your intentions

@Override

Public void displayRateOfPay () { }

**CALLING CONSTRUCTORS W/ INHERITANCE:**

* When instantiating an object that is a member of a subclass, you’re actually calling at least 2 constructors 🡪 base class constructor & extended class constructor
* Subclass object creation🡪 superclass constructor executes 🡪 subclass constructor executes
* When a superclass contains a default constructor 🡪 execution in subclass isn’t seen
* Where HourlyEmployee is a subclass of Employee:

HourlyEmployee clerk = new HourlyEmployee();

* both Employee() & HourlyEmployee() constructors execute
* When constructors initialize variables:
* superclass constructor should initialize data fields originating in the superclass
* subclass constructor only should initialize the data fields specific to the subclass

**Superclass Constructor Requiring Arguments:**

\*Default Constructor Review: Java supplies this when you create a class and don’t write one

(no arguments)

* When you write your own constructor you replace the automatically supplied version
* When a superclass only has constructors requiring arguments, all subclasses must provide the superclass with the needed arguments
* When **superclass has a default constructor** 🡪 can create subclass w/ or w/out its own constructor
* If subclass contains no constructor, all subclass objects use the superclass default constructor when they are instantiated
* When **superclass has only constructors requiring arguments** 🡪 each subclass must have at least 1 constructor
* The 1st statement w/in each subclass constructor must call a superclass constructor
* Must use keyword super to call superclass instructor from the subclass constructor:

\*Cannot use the superclass constructor name, MUST USE ‘SUPER’

* Syntax 🡪 super(list of arguments);

**public HourlyEmployee()** {

super(‘P’, 12.35, 40);

//any other statements

}

The version of the HourlyEmployee constructor requires no arguments, but passes 3 constant arguments to its superclass constructor

* This overloaded version of the HourlyEmployee constructor requires arguments:

**public HourlyEmployee(char dept, double rate, int hours)** {

super(dept, rate, hours);

//any other statements

}

**ACCESSING SUPERCLASS METHODS:**

* When a method has been overridden but you still want to use the superclass version w/ in the subclass 🡪 use the keyword super to access the parent class method

**COMPARING this AND super:**

\*In a subclass 🡪 keywords this & super sometimes refer to the same method, but sometimes do not

* For ex, when subclass overrides superclass method named someMethod()

🡪super.someMethod() refers to superclass version

🡪someMethod() & this.someMethod() refer to subclass version

* When subclass has not overridden the superclass method named someMethod()

🡪child can use the method name with

* super (because the method is a member of the superclass)
* this (because the superclass method is a member of the superclass by virtue of inheritance
* Alone (again, because the superclass method is a member of the subclass)

**INFORMATION HIDING:**

Public class Student

{

Private data fields //Can’t access/alter these w/out a public method

Public methods (get and set) //Access private data fields for the client

}

* When an application is a client of the Student class (that is, it instantiates a Student Object), the client cannot directly alter the data in any private field
* Main method:

Student someStudent = new Student();

SomeStudent.idNum = 812; // cannot access this field

someStudent.setIdNum(812); //must use the public set methods to access & alter

* Information Hiding: the concept of keeping data private
* Private members of the parent class are not accessible w/ in a child class’s methods
* \*Sometimes you want to access the parent class data w/ in the subclass
* Student 🡪 PartTimeStudent , FullTimeStudent
* If you want subclass methods to be able to directly access idNum and gpa🡪 can’t be private
* If you don’t want other nonchild classes to be able to access these data fields than they cannot be public
* Protected access provides an intermediate level of security between public and private access 🡪 allows access between classes and its descendants

**METHODS YOU CANNOT OVERRIDE:**

* STATIC:
* Subclass cannot override methods declared static in the superclass method
* Can hide a static method in the superclass by declaring a static method in the subclass w/ same signature as the static method in the superclass; this method that hides the superclass method cannot access the parent access using the super object
* You cannot refer to super in a static method
* A non-static method cannot override a static member of a parent class
* FINAL:
* Subclass cannot override methods that are declared final in the superclass
* You should use the final modifier with methods when you want every child class to use the original parent class version of the method
* Virtual method calls
* Inlining
* METHODS IN FINAL SUPERCLASS:
* When declaring a class to be final, all of its methods are final
* A final class cannot be a parent
* Cannot extend a final class

Video Lecture Notes – Ch 10 Inheritance

Interfaces:

* When a class implements an interface, that class must use (and define the implementation for) the methods in that interface
* When a class extends another class, that class has the option to use the methods in the parent class or not
* Declare methods in the interface class
* public interface InterfaceName
* Define method implementations in the class using the interface
* public class ClassName implements InterfaceName

IN THE CONTEXT OF THIS ARTICLE: <https://www.developer.com/lang/article.php/3642656/Designing-with-Interfaces-amp-Abstract-Classes.htm>

* You focus on using interfaces and abstract classes to create contracts and encourage reusable code
* You will consider a *contract* to be any mechanism that requires a developer to comply with the specifications of an Application Programming Interface (API). Often, an API is referred to as a framework
* In Java and the .NET languages, the two ways to implement contracts are to use abstract classes and interfaces

ABSTRACT CLASSES:

* **Abstract Class:** contains one or more abstract methods; can provide concrete methods as well
* **Abstract methods:** methods that do not have any implementation provided
* Subclasses must implement the abstract methods declared by the superclasses
* These abstract methods are the contract
* For example 🡪 an abstract class named Shape
* It is abstract because you cannot instantiate it
* If someone is asked to draw a shape, they’ll ask what kind of shape

🡪Therefore, the concept of a shape is abstract

**\*\*Example:** You’re creating an application to draw shapes 🡪 every shape in proposed design and any shapes that might be added later\*\*

* All shapes should use the same syntax to draw themselves:
* Every shape should contain a method called draw()
* Every class must be responsible for its own actions:
* Even though all of the classes must provide a method called draw(), each class must provide its own implementation of the code
* Creation of the subclasses Circle & Rectangle 🡪 each has its own version of draw()
* This shows a **Shape framework that is truly Polymorphic**
* The draw() method can be invoked for every shape in the system & invoking this method for each shape produces a different result
* Sending a message to an object evokes a different response, depending on the object

\*\*Code to illustrate how Rectangle and Circle conform to the Shape contract:

public abstract class Shape {

public abstract void draw();

}

* The class does not provide any implementation for *draw* ()
* The absence of code is what makes the method abstract

(providing code would make the method concrete)

public class Circle extends Shape {

public void draw() {System.out.println ("Draw a Circle");}

}

public class Rectangle extends Shape {

public void draw() { System.out.println ("Draw a Rectangle");}

}

* Both Circle and Rectangle extend (inherit from) Shape and provide their implementations of draw
* Here is where the contract comes in:
* If Circle inherits from Shape and fails to provide a *draw* () method, Circle won't even compile. Thus, Circle would fail in its attempt to satisfy the contract with Shape. A project manager can require that programmers creating shapes for the application must inherit from Shape. By doing this, all shapes in the application will have to provide a draw () method that complies with the Shape contract.
* Also,  If Circle does indeed fail to implement a *draw* () method, Circle will be considered abstract. Thus, yet another subclass must inherit from Circle and implement a *draw* () method. This subclass would then become the concrete implementation of both Shape and Circle.

\*\*Some languages, such as C++, use only abstract classes to implement contracts\*\*

* C++ supports multiple inheritance, Java & C# does not
* Java and C# classes can inherit from only one parent class, they can implement many **Interfaces:**

INTERFACES:

public interface Nameable {

String getName();

void setName (String aName);

}

* Nameable is declared as an Interface, not a Class
* Because of this, both methods are considered abstract and provide no implementation
* Any class that implements an interface must provide the implementation for all methods
* \*Java and C# 🡪 a class inherits from an abstract class whereas a class implements an interface

\*Both Abstract Classes and Interfaces provide Abstract Methods BUT:

* an Abstract Class provides both Abstract and Concrete Methods
* an Interface provides only Abstract Methods

**\*\*EXAMPLE:** design a class that represents a dog, with the intent of adding more mammals later

public abstract class Mammal {

public void generateHeat() {

System.out.println("Generate heat");

}

public abstract void makeNoise();

}

* Declared an abstract Class Mammal w/:
* A Concrete Method generateHeat() 🡪 every animal will generate heat
* An Abstract Method makeNoise() 🡪 every animal will make noise differently

public class Head {

String size;

public String getSize() {

return size;

}

public void setSize(String aSize) {size = aSize;};

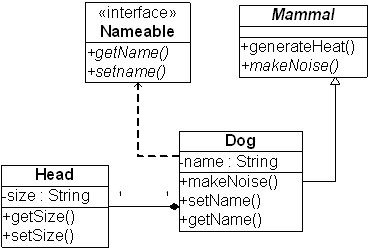
}

* This class Head is used in a composition relationship (A Dog has-a Head)
* It illustrates how composition relates to abstract classes and interfaces in the overall design of an object-oriented system

\*\* Remember that there are two ways to build object relationships: the **is-a** relationship, represented by inheritance; and the **has-a** relationship, represented by composition

**Where does Interface fit in??**

To tie everything together, create a class called Dog that is a subclass of Mammal, implements Nameable, and has a Head object



In a nutshell, Java and C# build objects in three ways: inheritance, interfaces, and composition. Note the dashed line in Figure 3 that represents the interface. This example illustrates when you should use each of these constructs. When do you choose an abstract class? When do you choose an interface? When do you choose composition? Explore this idea further.

* Dog is a Mammal, so the relationship is inheritance.
* Dog implements Nameable, so the relationship is an interface.
* Dog has a Head, so the relationship is composition.

public class Dog extends Mammal implements Nameable {

String name;

Head head;

public void makeNoise(){System.out.println("Bark");};

public void setName (String aName) {name = aName;};

public String getName () {return (name);};

}

**Links and Video Lectures**

**Java Programming Inheritance** **Tutorial**

<https://www.youtube.com/watch?v=9JpNY-XAseg>

**Java Polymorphism Tutorial**

<https://www.youtube.com/watch?v=SwEzCBM7n-Q>

**Learn Java Tutorial 1.10- Implements an Interface**

<https://www.youtube.com/watch?v=mCHcdAnXTtc>

**Designing with Interfaces & Abstract Classes**

<https://www.developer.com/lang/article.php/3642656/Designing-with-Interfaces-amp-Abstract-Classes.htm>