Java Chapter 11 – Advanced Inheritance

**INHERITANCE REVIEW:**

* When extending a class, the subclasses you create inherit all general attributes already defined on the base class
* Must only create the new, more specific attributes for the class

Employee: contains general employee attributes

SalariedEmployee & HourlyEmployee: contains all the general Employee attributes, as well as specific attributes like unique pay-calculating methods

* Superclass 🡪 contains the features shared by all of its subclasses
* Subclasses 🡪 more specific examples of the superclass type
* Child classes contain all the members of its parent 🡪 public, private, & protected
* BUT a child object cannot directly access private members of its parent

COMPARISONS:

Concrete class vs abstract class

Final class vs abstract class

**ABSTRACT CLASSES:**

\*You might never intend to create an object that’s “just” an Employee, and only want more specific objects\* 🡪 SalariedEmployee, HourlyEmployee, ContractEmployee;

🡪Therefore, Employee is only created to be extended/inherited from

* Abstract Class: only created to be extended/inherited from
* Cannot create any concrete objects of an abstract class
* Usually contains at least 1 abstract method
* Can include 2 method types:
* **Non-Abstract Methods:** (Concrete?) Implemented in the abstract class and are simply inherited by its children; like methods you create in any class
* **Abstract Methods:** have no body and are only declared in the abstract parent class;

🡪Must be implemented in the child classes

\*creating an empty method w/in an abstract class 🡪 the method is automatically abstract even w/out using the keyword

* When making abstract declarations:
* Classes 🡪 each of its methods can be abstract or not
* Methods 🡪 must declare class as abstract if it contains any abstract methods
* When a subclass inherits an abstract method:
* You’re required to code a subclass method to override every empty/ abstract superclass method it inherits
* Must provide ‘{ }’ body/implementation for the inherited abstract method

OR the child class method must be abstract itself

**public abstract class Animal** //abstract Animal class

{

private string name;

public string getName() {

return name;

}

public void setName(String animalName) {

name = animalName;

}

}

**public class Dog extends Animal** //Dog class

{

@Override

public void speak() {

System.out.println(“Woof!”);

}

}

**Public class Cat extends Animal** //Cat class

{

@Override

public void speak() {

System.out.println(“Meow!”);

}

}

**Public class UseAnimals** //UseAnimals Application

{

Public static void main(String[] args) {

Dog myDog = new Dog;

Cat myCat = new Cat;

MyDog.setName(“Spot”);

myCat.setName(“Oscar”);

System.out.print(myDog.getName() + “ says ”);

myDog.speak();

System.out.print(myCat.getName() + “ says ”);

myCat.speak();

}

}

\*Polymorphism 🡪 using the same method name to indicate different implementations

🡪one method name causes different & appropriate actions for diverse types of objects

**DYNAMIC METHOD BINDING:**

* When you create a superclass & 1+ subclasses, each object of each subclass “is a” superclass object

SalariedEmployee “is an” Employee & Dog “is an” Animal

* Therefore, you can convert subclass objects into superclass objects
* Abstract Superclass 🡪 cannot instantiate an objects of this type
* BUT you can indirectly create a reference to a Superclass Abstract Object
* \*a Reference is not an object but it points to a memory address of a concrete object
* Creating a Reference:
* Do not use the keyword ‘new’ to create a concrete object
* INSTEAD create a variable name to hold the memory address of a concrete object

**public class AnimalReference**

{

public static void main(String[] args)

{

Animal animalRef;

animalRef = new Dog();

animalRef.speak();

animalRef = new Cat();

animalRef.speak();

}

}

* This creates a generic Animal Reference Variable into which you can assign any of the Concrete Animal Child Objects
* Assigning a variable or constant of one type to a variable of another type 🡪 upcasting
* Using a reference polymorphically allows you to extend a base class & use extended objects when a base class type is expected

Ex: can pass a Dog or Cat to a method expecting an Animal type

* All methods written to accept a superclass argument can also be used w/ its subclasses
* Each reference “chooses” the correct speak() method based on the type of animal referenced
* This behavior is most useful when you pass references to methods

\*Inheritance Review\*:

* In Java, all instance method calls are Virtual Method Calls by default – The method that is used is determined when the program runs, because the type of the object used might not be known until the method executes
* Dynamic Method Binding:

\*Instance methods (those that receive a ‘this’ reference) use this\*

* An applications ability to select the correct subclass method based on the argument type
* When the app executes, the correct method is attached (bound) to the application based on the current, changing (Dynamic) context
* Makes programs more flexible
* Static (fixed) Method Binding:

\*Class methods use this\*

Using a Superclass as a Method Parameter Type:

* Dynamic method binding is most useful when you want to create a method that has 1+ parameters that might be one of several types

**Public class TalkingAnimalDemo**

{

Public static void main(String[] args)

{

Dog dog = new Dog();

Cat cat = new Cat();

dog.setName(“Spot”);

cat.setName(“Oscar”);

talkingAnimal(dog);

talkingAnimal(cat);

}

Public static void talkingAnimal(Animal animal) {

System.out.println(animal.getName() + “ says ” animal.speak());

}

}

* The method can be used in programs that contain objects of any class, as long as they’re descendants from Animal & will work no matter which type of Animal it receives

\*Array Review\*:

* Every element must be the same data type 🡪primitive, built-in, etc.
* When creating Arrays 🡪 you are not constructing any Objects
* You are creating space for references to Objects
* An “Array of Objects” really means an “Array of Object References”

**ARRAYS OF SUBCLASS OBJECTS:**

* When creating an array of superclass references, the array can also hold subclass references

\*This is true whether the superclass is Concrete or Abstract\*

* Ex. Employee Array: holds SalariedEmployee & HourlyEmployee

Animal [] animalRef = new Animal[2];

* This reserves memory for 2 Animal Objects / Object References
* This does not instantiate any Animals (Animal 🡪 abstract 🡪 can’t create objects anyways)

**public class AnimalArrayDemo**

{

public static void main(String[] args)

{

Animal [] animalRef = new Animal[2];

animalRef[0] = new Dog();

animalRef[1] = new Cat();

for(int x = 0; x < animalRef.length; ++x) {

animalRef[x].speak();

}

}

}

* A Reference to an instance of the Dog class is assigned to the 1st Animal reference
* A Reference to a Cat Object is assigned to the 2nd array element
* For loop is used to get each individual reference to speak()

**OBJECT CLASS:**

* When you don’t explicitly extend another class, your created class is implicitly an extension of the Object Class
* Public class Animal{} & public class Animal extends Object{}

🡪 same results

* Object Class contains methods that descendants can use, **overload**, or override

**toString() Method:**

* Converts an object into a String that contains info about the object
* \*Automatic Object Class version: returns🡪 class name of which the object is an instance of, the @ sign, & a hexadecimal identifier (hash code)
* \*More useful to write an overloaded version that displays some/all data field values of the object
* This is useful when debugging to pinpoint errors
* **@Override**

**public String toString()** {

String info = //info to be displayed;

return info;

}

\*Could also use get methods to achieve these results, BUT toString() is Java’s conventional name for a method that converts an objects relevant details/info into string format

\*Side Note: the toString() method in the String class converts any object into a string

🡪the String class overloads the Object class’s version to achieve this

**equals() Method:**

**\*public boolean equals(Object obj)**

* Is not static, & takes a single argument that is compared to the calling object
* Returns a bool value indicating whether the objects are equal
* Considers 2 objects to be equal only if they have the same hash code
* Are equal only if one object is a reference to the other

**public class Compare Accounts //BankAccount class below**

{

public static void main(String[] args)

{

BankAccount acc1 = new BankAccount(1234, 5000.00);

BankAccount acc2 = new BankAccount(1234, 5000.00);

if(acc1.equals(acc2))

System.out.println(“Accounts are equal”);

else

System.out.println(“Accounts are not equal”);

}

} //this will return ‘Accounts are not equal’ even though their values are equal

* To compare objects based on the values they hold:
* Create your own implementation from scratch
* **Overload the Object class equals() method 🡪 easiest**
* Override the Object class equals() method

**Overloading equals():**

* Write your own comparison method in the class w/ a different parameter list

**public class BankAccount**

{

private in accNum;

private double balance;

public BankAccount(int num, double bal) {

accNum = num;

balance = bal;

}

@Override

public String toString() {

String info = “BankAccount accNum = ” + accNum + “ Balance = $” + balance;

return info;

}

**public boolean equals(BankAccount secondAcct)** {

boolean result;

if(accNum == secondAcct.accNum && balance == secondAcct.balance)

result = true;

else

result = false;

return result;

}

}

* The object that calls the method is held by the ‘this’ reference w/in the method

🡪accNum & balance refer to acct1 object values

acct2 is the argument 🡪 secondAcct refers to acct2

* BankAccount object now has access to 2 equals methods 🡪 1 that takes a BankAccount parameter & 1 that takes an Object parameter;

🡪BankAccount equals() method overloads the Object equals() method version

**Overriding equals():**

\*When a subclass method overrides a parent method, the signatures must be the same\*

* Method header must match ‘public boolean equals(Object obj)’w/ any identifier for the Object parameter
* With this signature 🡪 must cast the Object parameter to a BankAccount object before comparing:

casting statement 🡪 BankAccount secondAcct = (BankAccount)obj;

* \*\*\*Recommendations/Rules for overriding equals()\*\*\*

USING INHERITANCE TO ACHIEVE GOOD SOFTWARE DESIGN:

**INTERFACES:**

\*Some object-oriented programming languages allow a subclass to inherit from more than 1 parent class\*

* Multiple inheritance is prohibited in Java
* Single inheritance w/ multiple generations is allowed
* Interface is an alternative to multiple inheritance; is similar to a class BUT also:
* All methods are implicitly public and abstract
* All data items are implicitly public, static, & final
* It’s a description of what a class does, but now how it is done

🡪declares method headers, but not the instructions w/in those methods

* A class that uses an interface will use the keyword ‘implements’
* Implements: requires a subclass to implement its own version of each method
* Extends: allows subclass to use nonprivate, nonoverridden members of its parent class

**public interface worker**

{

public void work();

}

* When any class implements Worker, its must either include a work() method or the new class must be declared abstract, & then its descendants must implement the method
* If many classes will be Workers (require a work() method) they can all implement work()
* When a class implements an interface it represents a situation similar to inheritance

Ex: a WorkingDog (a class that implements the Worker Interface):

**🡪“is a”** dog, & **“is an”** Animal, & also **“is a”** Worker

* Comparing Abstract Classes & Interfaces:
* Abstract classes and interfaces are similar in that you cannot instantiate concrete objects from either one
* Abstract classes differ from interfaces because abstract classes can contain non abstract methods, but all methods w/in an interface must be abstract
* A class can inherit from only one abstract superclass, but can implement many interfaces
* When to use:
* Abstract class 🡪 when you want to provide data/methods that subclasses can inherit, but at the same time these subclasses maintain the ability to override the inherited methods
* Interface 🡪 when you know what actions you want to include, but you also want every user to separately define the behavior that must occur when the method executes;

&/or when you want a class to implement behavior from 1+ parent

* Interfaces for storing related Constants:
* Provides a set of data that a number of classes can use without having to redeclare the values

**PACKAGES:**

* A named collection of classes; ex: java.lang
* When you create classes, you can place them in packages so that you or other programmers can easily import your related classes into new programs
* When creating professional classes for others to use 🡪
* don’t provide source code files w/.java extension
* provide users w/ the compiled files w/ the .class extensions
* Default Package: where a class is placed by default if you don’t specify otherwise
* A class placed into a nondefault package must be public
* If a class isn’t public, it can only be used by classes w/in the same package
* Package Declaration: places a class in a particular package; indicates folder / pathname
* Must be the first statement in the file
* Import declarations follow the package declaration
* Can copy or move the compiled class file to the appropriate folder OR
* Can use a compiler option w/ the javac command 🡪 -d
* Package Naming Convention:
* Uses your internet domain name in reverse order

🡪If domain is ‘course.com’, begin all package name w/ ‘com.course’

**Links and Video Lectures**

**Inheritance in Java**

<https://beginnersbook.com/2013/03/inheritance-in-java/>

<https://www.tutorialspoint.com/java/java_inheritance.htm>

**Inheritance in Java Video**

<https://www.youtube.com/watch?v=i62tHwVnoE8>