**Week 1 – Introduction to Object-Oriented Concepts**

**Learning Outcomes**

By the end of this week, you will be able to

* define object-oriented programming
* distinguish between an object and a class
* describe how object-oriented programming is fundamentally different from the programming you have done before.
* Justify the choice to use an object-oriented approach to developing software.
* Identify and define six object-oriented concepts: abstraction, encapsulation, information hiding, ownership, inheritance, and polymorphism
* Distinguish between aggregation and composition as two different kinds of ownership.
* Identify the superclass and the subclass in an inheritance relationship.
* Demonstrate inheritance, ownership, and polymorphism in snippets of Java code
* Depict classes and their relationships using UML class diagrams.
* Define and demonstrate the software patterns delegation, singleton, factory, and model-view-controller
* Explain through examples the following software design principles: Open-Closed, Dependency-Inversion, Interface Segregation, Liskov Substitution, and Single Responsibility.
* Identify and describe characteristics of bad software: rigidity, immobility, fragility

**What is object-oriented programming?**

It’s programming with objects.

**What’s an object?**

An instance of a class

**What’s a class?**

It’s a data type that specifies both data and functions that operate on or with that data. In other words, it’s a prototype or blueprint for building something that has both knowledge and skills.

**How is this different from programming you’ve done before?**

You have to think in terms of nouns – actors – rather than in terms of verbs – actions.

**Why?**

Object-oriented programming brings a whole bunch of neat concepts to the table that enable us to build sophisticated software solutions more easily than we otherwise would have been able to.

**Object-Oriented Programming has …**

* Concepts – powerful features that prove indispensible to modern software development, brought to us automatically by object-oriented programming.
* Patterns – tried-and-true templates for forging relationships between classes
* Principles – guidelines that help you determine what classes are needed and how they should divide up the work

**What are the main object-oriented programming concepts?**

Encapsulation

Information Hiding

Aggregation and Composition (two kinds of ownership, with aggregation being shared ownership and composition being exclusive ownership)

Inheritance

Polymorphism

**Encapsulation**

Here is an example of encapsulation – bundling data and methods together in one data structure.

public class Student {

private String studentID;

private Date dob;

private String address;

public String getID() {

return studentID;

}

public void setID(String id) {

studentID = id;

}

public void takeCourse() {

}

public void earnGrade() {

}

public void fileToGraduate() {

}

}

...

Student favorite = new Student();

favorite.setID("867530900");

...

System.out.println("The id is " + favorite.getID());

**Inheritance**

Here is an easy-to-understand example of inheritance:

class Shape {

}

class Circle extends Shape {

}

class Rectangle extends Shape {

}

Two terms related to inheritance:

* superclass – the parent
* subclass – the descendant

In this example, Shape is the superclass. Circle and Rectangle are subclasses.

**Abstraction**

Abstraction is another key concept. Something is abstract when it is a concept but is not concrete or defined enough to actually be built. Generally, in OO design, we start with abstract things, and then we build on them through inheritance.

An *abstract class* is one that has one or more *abstract methods*.

An *abstract method* is a method / function that has no body – just a name, return type, and parameters.

An *interface* is the strictest interpretation of an abstract class – it is a data structure that consists entirely of abstract methods. In other words, none of its methods/functions have a body.

Abstraction is related to inheritance. Often, when we construct families of related objects, we start the family with an abstract class that represents the least common denominator for everyone in that family. In other words, what do all classes that are part of that family have in common? We often (not always, but often) put that in an abstract class.

**Polymorphism**

Polymorphism enables you to process collections of related things generically. This is particularly useful when you want to use a loop to march through a collection of items.

Example of polymorphism: a for loop that moves through the entries of a list. The list might be of a collection of related kinds of object. We can refer to each of the objects in the list through a generic variable (whose data type matches the one that all are ultimately related to). But, when we invoke a particular function that all members of the family share, each will respond by performing that function in their own specific way. For example, we could have a collection of Shape objects. We could refer to each entry in the Shape list through a generic Shape variable, even though the actual entries in the list are specific kinds of shapes – Circle, Rectangle, etc. All Shape objects might have the ability to calculate their own area. When we refer to an object in the list through a generic Shape variable and tell it to calculate its area, thanks to polymorphism, the circle version of the area() function will be called when we’re dealing with a circle, and the Rectangle version of area() will be called when we’re dealing with a rectangle, etc.

Shape[] shapes = new Shape[3];

shapes[0] = new Circle();

shapes[1] = new Rectangle();

shapes[2] = new Triangle();

for (Shape s : shapes) {

System.out.println(s.area());

}

The first time through the for loop, we’ll call the Circle.area() function – actually, we won’t; it will happen automatically. The next time through, we’ll call the Rectangle version, and then we’ll call the Triangle version.

Polymorphism is implemented behind the scenes using a Virtual Method Table (VMT). The VMT keeps track of where various related classes’ same-named functions are located in memory. Using the VMT, the operating system is able to figure out which code to implement when we tell each shape to fire its area() function.

**Composition**

An example of composition: the relationship between Face and Nose, Mouth, and Eye

class Face {

Nose n;

Mouth m;

Eye le;

Eye re;

}

We probably wouldn’t let the Nose, Mouth, or Eye objects live beyond the Face. They are owned exclusively by the Face. That’s what composition is: exclusive ownership.

One clear sign that we are dealing with composition is if the owner (Face, for example) is responsible for actually creating the objects it owns. Then, clearly, the things that are owned – the Nose, Mouth, etc. – could not have existed on their own and are therefore exclusively owned by the owner object.

**Aggregation**

Aggregation is also a form of ownership, but it’s non-exclusive ownership. The owned objects can live on and perhaps existed prior to the owned object.

An example of aggregation: A library patron borrows a book. That’s not exclusive ownership, since several people can borrow a book over its lifetime.

The easy way to tell if something is composition or aggregation is to ask if the owner is responsible for creating and destroying the thing that is owned. If so, it’s a composition relationship.

**UML**

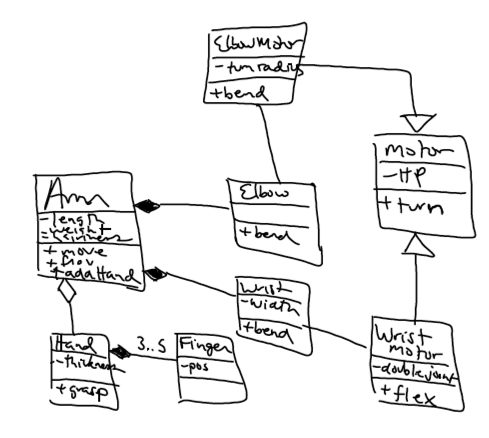
UML, or Unified Modeling Language, is a way to show a system’s architecture in graphical form. UML represents

* classes as boxes with three sections, the top of which specifies the name of the class, the middle of which specifies the data, and the bottom of which specifies the functions.
* Lines between the classes.
  + A line with an arrow / triangle pointing to the parent – inheritance
  + A line with a filled diamond next to the owner – composition
  + A line with an open diamond next to the owner – aggregation
  + A line with no decorations – just an association (a using kind of relationship)

UML helps us see the design without having to wade through lots of text to understand it.

**Some UML Examples**

Robot Arm



*Corresponding Code:*

class Arm {

private Elbow elb;

private Wrist wri;

private Hand hnd;

public Arm() {

elb = new Elbow();

wri = new Wrist();

hnd = null;

}

public void addHand(Hand h) {

hnd = h;

}

}

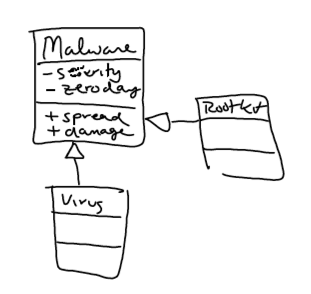
...

Arm arm = new Arm();

Hand hand = new Hand();

arm.addHand(hand);

**Another Example - Malware**



*Corresponding code:*

abstract class Malware {

private int severity;

private boolean zeroDay;

public abstract void damage();

public abstract void spread();

}

class Virus extends Malware {

public void damage() {

}

public void spread() {

}

}

**UML Example:** Draw the diagram for this code.

class NtwkDev {

}

class Router extends NtwkDev {

}

class Switch extends NtwkDev {

}

class Firewall extends NetwkDev {

}

class OS {

}

class Server {

private OS opsys;

}

class Network {

Router[] routers;

Firewall[] firewalls;

Switch[] switches;

Server[] servers;

public Network() {

routers = new Router[100];

switches = new Switch[100];

firewalls = new Firewall[100];

servers = new Server[100];

**Object-Oriented Programming Also Employs Patterns**

A software design pattern is a commonly repeated approach to constructing software. These approaches are commonly repeated because they have found to be useful.

Some common design patterns include:

***Singleton*** (making sure there is just one instance of something to avoid conflicts - <http://www.oodesign.com/singleton-pattern.html>) These are great for coordinating activity across multiple threads of execution.

A singleton can be written by making the constructor for the class private and equipping the class with a static function that ensures that the constructor is called only if no other objects of that class already exist.

**Singleton Example**

class Widget {

public static Widget theOne = null;

public static Widget buildWidget {

if (theOne == null) {

theOne = new Widget();

}

return theOne;

}

private Widget() {

// builds a widget

}

}

Widget variable = Widget.buildWidget();

***Factory*** (create an object without exposing how it is created - <http://www.oodesign.com/factory-pattern.html>)

An example of a Factory pattern would be a BeastFactory that builds enemies of various kinds in a video game. There might be 50 varieties of beasts. Rather than force fellow programmers to become intimately familiar with the details of creating each different kind, we could have central point responsible for building beasts – the BeastFactory, and equip it with a function called buildBeast that takes in a string that describes the kind of beast we wish to build.

**Factory Example**

abstract class Shape {

public abstract double findArea();

public abstract double findPerim();

}

class Circle {

public Circle(int x, int y, int rad) {

}

public double findArea() {

}

public double findPerim() {

}

}

class Rectangle

public Rectangle(int x, int y, int w, int h) {

}

public double findArea() {

}

public double findPerim() {

}

}

class ShapeFactory {

static Shape createShape(String params) {

// split params into parts

int x, y, rad, w, h;

if (parts[0].equals("circle")) {

x = Integer.parseInt(parts[1]);

y = Integer.parseInt(parts[2]);

rad = Integer.parseInt(parts[3]);

return new Circle(x, y, rad);

} else if (parts[0].equals("rectangle")) {

//extract x, y, w, and h from parts

return new Rectangle(x,y,w,h);

} etc.

}

}

Rectangle myRectangle = ShapeFactory.createShape("rectangle 5 10 15 20");

***Delegation*** (chain of command; ask an object to do something, which tells something else to do that thing)

Example:

class OrganizationTeam {

public void advance() {

// do something

}

}

class Organization {

OrganizationTeam marketing;

OrganizationTeam engineering;

public void advance() {

marketing.advance();

engineering.advance();

}

}

**Another Example of the Delegation pattern**

In this example, we have software for managing inventory for a manufacturer who makes classroom furniture. We have a variety of writing surfaces that consist of parts. We want to print out our inventory of parts.

class Part {

}

class WritingSurface {

private Part[] parts;

public Part[] listParts() {

}

}

class Podium extends WritingSurface {

}

class StudentDesk extends WritingSurface {

}

class PartsInventory {

// list all the components that comprise a factory's models

WritingSurface[] models;

public Part[] listParts() {

for (WritingSurface ws : models) {

ws.listParts();

}

}

}

PartsInventory pi = new PartsInventory();

pi.listParts();

***Model-View-Controller*** – our primary focus this semester. It guides us to separate model – the data – from how we view the data – the view – using an intermediary called the controller.

Example of Model-View-Controller: Suppose we have a system that manages student data. The Student class would be the model; a StudentEntryForm could be the view, and a FillStudentEntryFormController could be the controller class responsible for populating a student data entry form with data.

class Student { //model

private String studentID;

private String lastName;

private String firstName;

}

class StudentDataEntryForm extends JFrame { //view

JTextField txtLastName;

JTextField txtFirstName;

JTextField txtStudentID;

public void fillFields(String sid, String lastName, String firstName) {

}

}

class FillDataEntryController

public void fillStudentForm(Student s, StudentDataEntryForm sdef) {

sdef.fillFields(s.getStudentID(), s.getLastName(),

s.getFirstName());

}

}

**Why Use Patterns?**

Using patterns helps us write good software more regularly, because they are tried-and-true approaches to writing it.

**What are the qualities of bad software?**

These were expressed by Robert Martin in *Agile Software Development: Principles, Patterns, and Practices:*

*Rigidity*: hard to change the software because changes impact other parts

*Fragility*: when you make a change, unexpected other parts break.

*Immobility*: you can’t disentangle functionality to use it elsewhere.

**Good software**

Doesn’t have these problems.

**Software design is also governed by a number of software design principles that help improve the quality of software.**

* Open-Close Principle (OCP) – software entities should be open for extension, not for modification. (http://joelabrahamsson.com/a-simple-example-of-the-openclosed-principle/. This example also demonstrates the power of inheritance and polymorphism.)
* Dependency-Inversion Principle – don’t let high-level classes or modules depend directly on low-level ones, because then they’ll be tightly coupled, and that would increase the rigidity, fragility, and immobility. (Example – Manager – Worker relationship described at http://www.oodesign.com/dependency-inversion-principle.html)
* Interface segregation Principle – don’t fatten up your modules with stuff they don’t need. (Example – Workers vs. robot workers, described at http://www.oodesign.com/interface-segregation-principle.html)
* Single Responsibility Principle – a class should have only one reason to change. (example – an email class with both content and protocol – see http://www.oodesign.com/single-responsibility-principle.html)
* Liskov’s Substitution Principle – an object of a derived class should be completely substitutable for objects of its base class (square vs. rectangle example: http://www.oodesign.com/liskov-s-substitution-principle.html)

**Open-Closed Principle Example**

Software should open to extension and closed to modification. You shouldn't have to change the code to have it handle new, related circumstances.

**Bad:**

class AreaCalculator {

public double calcArea(Shape s) {

if (s instanceof Circle) {

Circle c = (Circle)s;

return c.radius \* c.radius \* Math.PI;

} else if (s instanceof Rectangle) {

Rectangle r = (Rectangle)s;

return r.width \* r.height;

} else if (s instanceof Triangle) {

Triangle t = (Triangle)s;

return 0.5 \* t.base \* t.height;

}

}

}

**Good:**

abstract class Shape {

public abstract double calcArea();

}

class Circle extends Shape {

private double radius;

public double calcArea() {

return radius \* radius \* Math.PI;

}

}

class Rectangle extends Shape {

private double w;

private double h;

public double calcArea() {

return w \* h;

}

}

class Triangle extends Shape {

public double calcArea() {

}

}

class AreaCalculator {

public double calcArea(Shape s) {

return s.calcArea();

}

}

**Dependency Inversion Principle**

Don't let owners depend on the implementation of the things it owns.

Example: Manager and Worker

**Bad: Managers have to distinguish and handle separately two different kinds of workers**

class Manager {

Worker[] workers;

SuperWorker[] sworkers;

public void doWork() {

for (Worker w : workers) {

w.doWork();

}

for (SuperWorker sw : sworkers) {

sw.doWork();

}

}

}

One way to comply with Dependency Inversion Principle is to use an **interface**.

An interface is a class-like data type that prescribes behaviors rather than

data.

An interface a set of abstract functions.

Classes then \*implement\* that interface. In other words, they implement all

of the function that make up the interface.

Whereas you can you extend only one class, you can implement as many interfaces as you need.

interface IWorker {

public abstract void doWork();

public abstract void eatLunch();

}

class Worker implements IWorker {

public void doWork() {

}

public void eatLunch() {

}

}

class SuperWorker implements IWorker {

public void doWork() {

}

public void eatLunch() {

}

}

**Good: Managers now don’t have to differentiate between regular workers and super workers.**

class Manager {

IWorker[] workers;

public void doWork() {

for (IWorker w : workers) {

w.doWork();

}

}

}

**Interface Segregation Principle**

The example we just did has a teensy flaw. Consider the IWorker interface. It specifies that workers both eat lunch and do work. What if we end up building and using robotic workers? They don’t have to eat. So, our definition of Worker includes too much and therefore can only be clumsily applied to situations where our understanding of what a worker is might change.

**NOT IDEAL:**

interface IWorker {

public abstract void eat();

public abstract void work();

}

public class Robot implements IWorker {

// implement work() and, somehow, eat()

}

**BETTER:**

interface IWorkable {

public abstract void work();

}

interface IFeedable {

public abstract void eat();

}

class Robot implements IWorkable {

// implement work()

}

class Employee implements IWorkable, IFeedable {

//implement both work() and eat()

}

**Single Responsibility Principle**

A class should have only one reason to change.

An example might be a video player. If you build into the video player class both the responsibility to render the content and control the networking and buffering, you now have two reasons to have to change that VideoPlayer class. Instead, separate out the rendering and the networking/buffering among classes that the VideoPlayer class owns.

**Liskov Substitution Principle**

Subclasses should not break core functionality introduced by their parent.

For example: this is bad, because Square is not just setting its width in the setWidth function; it is also setting its height:

class Rectangle {

protected int w, h;

public double calcArea() {

return w \* h;

}

public void setWidth(int wid) {

w = wid;

}

}

class Square : Rectangle {

public void setWidth(int wid, int ht) {

w = wid;

**h = ht;**

}

}

Not cool!

**Bottom Line About These Principles**

We’ll see these principles again as the course continues.

I know this is somewhat heady theory at this point, but it will make more sense as we put these concepts into practice.

**This course mixes theory and practice**

We will learn and re-learn these concepts as we learn three object-oriented programming languages: Python, Java, and hopefully a little C# at the end.