Object Oriented Design Principles

Yulia Newton, Ph.D.

CS151, Object Oriented Programming

San Jose State University

Spring 2020

The Java Object Model

- Object model in Java is defined with these concepts
  - Class/interface, inheritance, encapsulation, polymorphism, message passing
- Design pattern with data centric classes with encapsulation in mind
- Data and data validation are performed in the same class
- Everything in Java is a class
  - Object class is the superclass of every class in Java
  - Defined in *java.lang* package

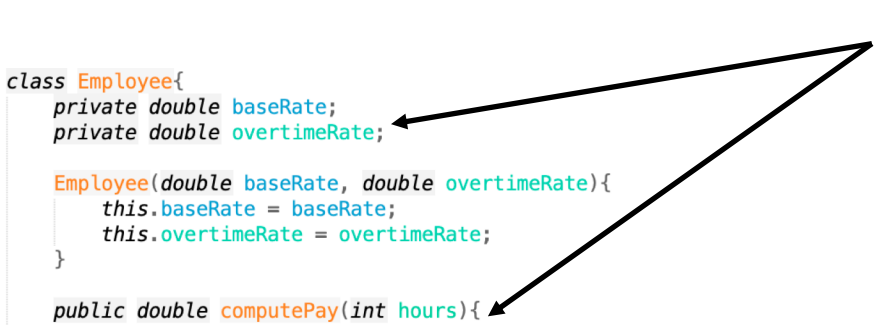
Core principles of OOD

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation

- Idea of hiding data and implementation from the users
- Combine related functions and variables into a single unit (object)
  - Functions = methods
  - Variables = properties
- Separating interface from implementation
  - Your class is a black box
    - Functionality implementation is inside the box
    - Users only see public declarations
- Encapsulation
  - Reduces data dependency
  - Increases flexibility (code changes can be made without affecting other parts of the program)
  - Increases reusability
  - Gives better control of data and methods (better security)

Example: Employee class



Pay rate fields and the logic of how they are used are hidden from the users of this class

```
class Employee{
    private double baseRate;
    private double overtimeRate;

    Employee(double baseRate, double overtimeRate){
        this.baseRate = baseRate;
        this.overtimeRate = overtimeRate;
    }

    public double computePay(int hours){
        if(hours > 40){ return(40 * this.baseRate + (hours - 40) * this.overtimeRate); }
        else { return(hours * this.baseRate); }
    }
}

public class test {
    public static void main(String args[])
    {
        Employee e1 = new Employee(15, 25);

        System.out.println("Employee works 20 hours and makes $"+e1.computePay(20));
        System.out.println("Employee works 50 hours and makes $"+e1.computePay(50));
    }
}
```

```
Employee works 20 hours and makes $300.0
Employee works 50 hours and makes $850.0
```

Good encapsulation practices

- Declare class variables/data private
- Provide setters and getters
  - Make sure setters modify variables to valid states
- Nested classes help increase encapsulation
  - Remember to use caution when nesting classes – weigh your options

Abstraction

- Making coffee with a coffee machine
  - You need to know the basic instructions
    - Need water, coffee beans, push the switch on
  - You do not need to know how the coffee machine works internally

- In programming world
  - Simpler interface
    - Understanding an object with a few properties and methods
  - Reduced impact of code change
    - Changing internal code and private methods does not change the interface
  - Hiding some details and showing only essential information

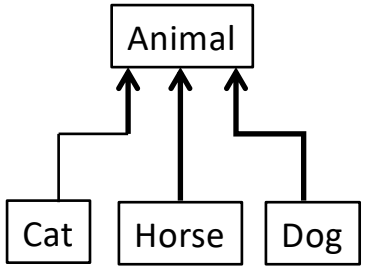
Abstraction in Java

- Achieved through
  - Abstract classes
  - Interfaces
- Both classes and methods can be declared as abstract
  - Cannot instantiate abstract classes
  - Have to implement abstract methods in children

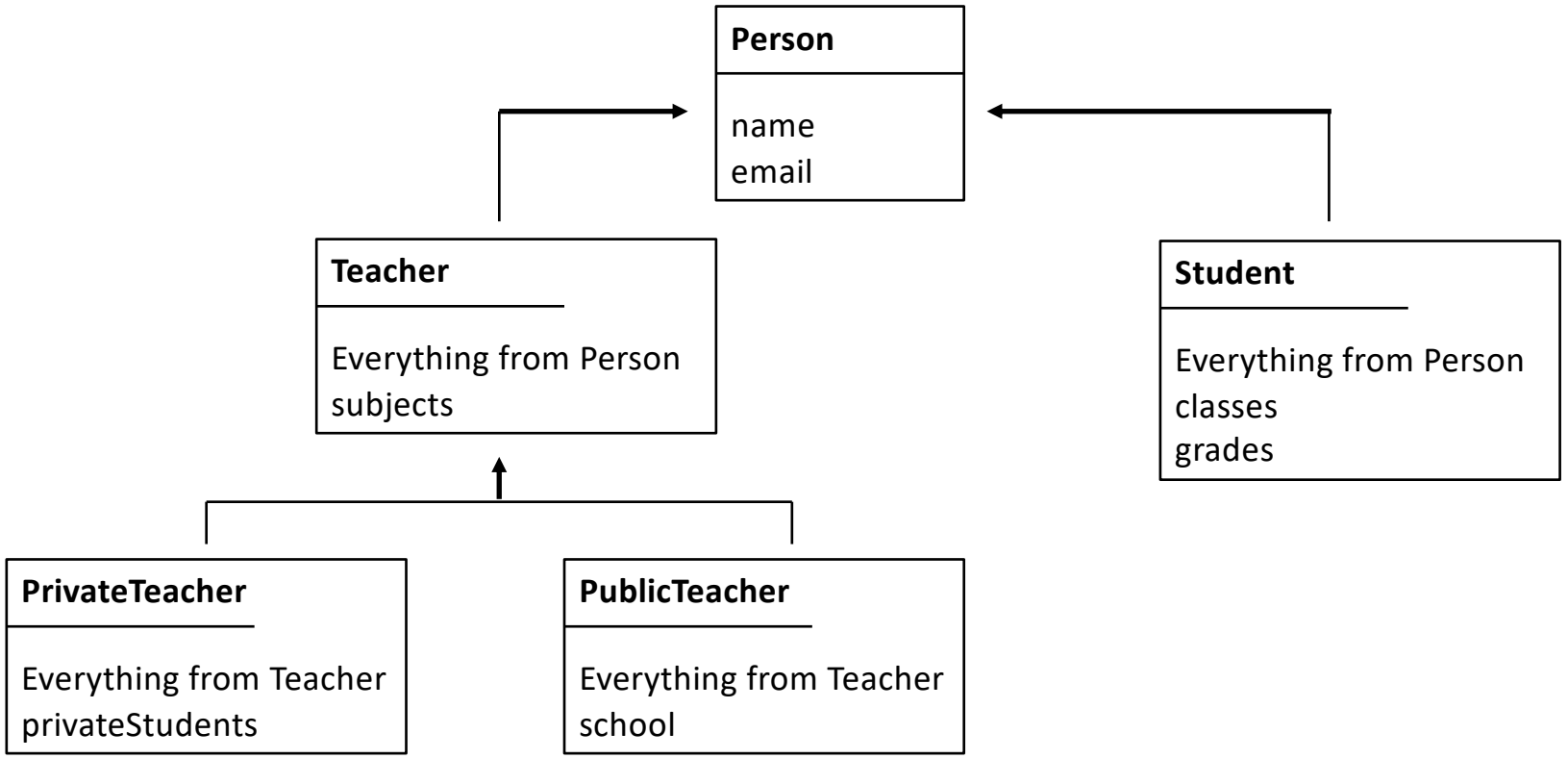
```java
abstract class Animal {
  public abstract void makeSound();

  public void introduce()
  {
    System.out.println("I am animal");
  }
}
```

Inheritance

- Helps eliminate redundant code
- Child objects inherit public properties and methods of parent objects
- Allows defining a new class in terms of another class



Example: Person class inheritance



Inheritance in Java

- Child class inherits all public methods and attributes of the parent class
  - We often call child class a *subclass*
  - We often call parent class a *superclass*
- Use keyword *super* to access the parent object
- Achieve multiple inheritance by using interfaces
  - Use keyword *extends* for classes
  - Use keyword *implements* for interfaces
- *final* class stops inheritance chain

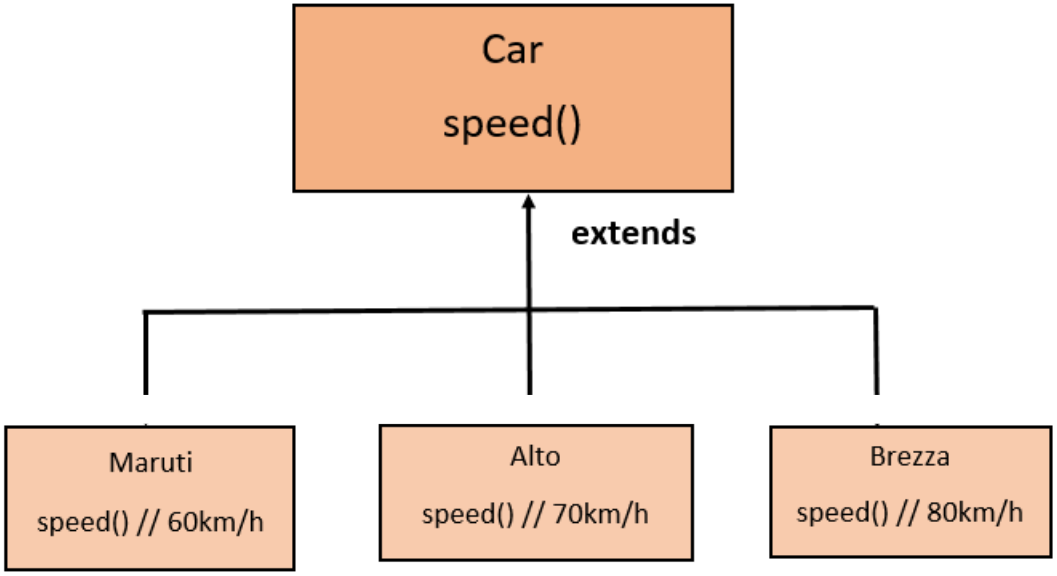
Polymorphism

- Many forms
  - Ability to do the same thing in multiple ways
- The same method behaves differently on the type of object that it is being used with
  - Perform single action in many ways

```
public class Horse extends Animal{
        ...
        public void sound(){
                System.out.println("Neigh");
        }
}
```

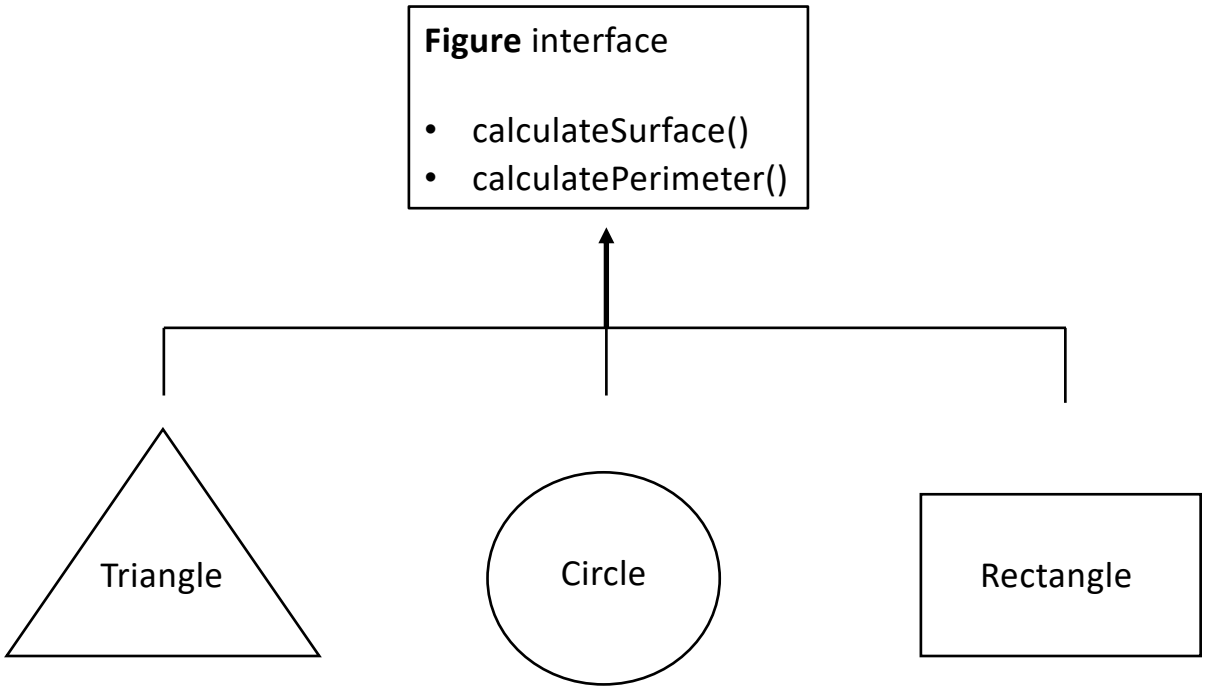
```
public class Cat extends Animal{
        ...
        public void sound(){
                System.out.println(”Meow");
        }
}
```

Another example of polymorphism



https://www.javastudypoint.com

Yet another example of polymorphism

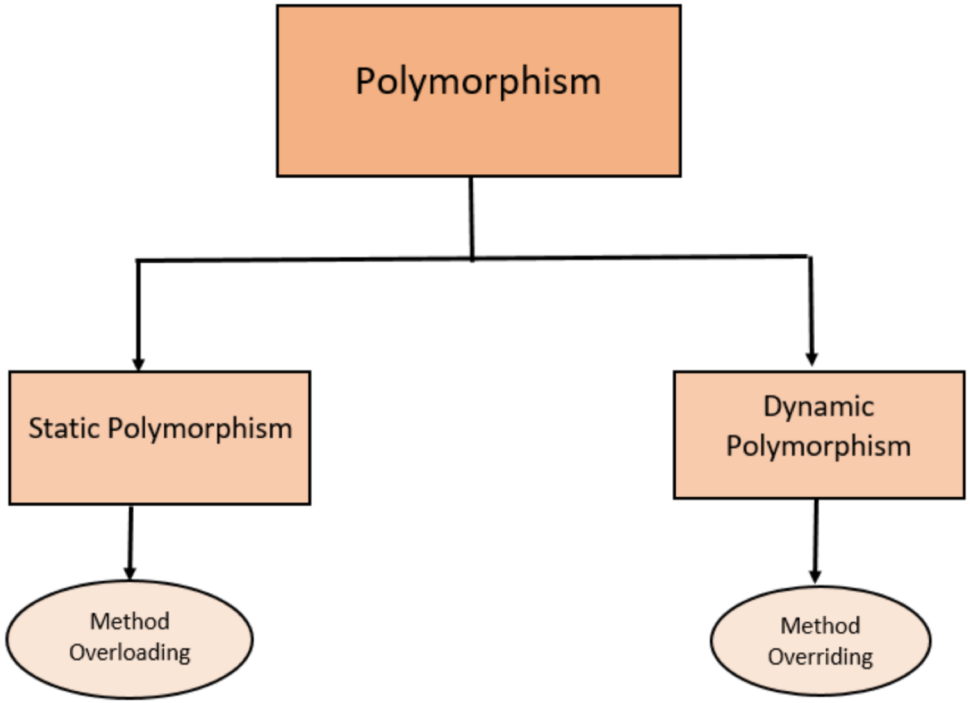


Inheritance vs. polymorphism

- Inheritance allows us to use parent's public methods
- Polymorphism allows us to use them in a different way than parents

Types of polymorphism in Java

- Static polymorphism
  - Compile time polymorphism
- Dynamic polymorphism
  - Run time polymorphism



https://www.javastudypoint.com

Static polymorphism (method overloading)

- Methods in a class have the same name but different number of parameters
  - Constructor overloading

- Ways to overload methods
  - Change the number of parameters
  - Change parameter data types
  - Change the order of parameters/data types

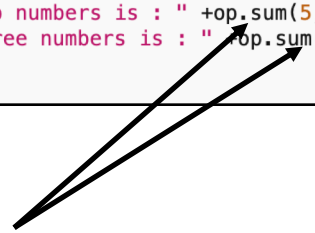
Example: changing number of parameters

Without method overloading:

```
class Operations{
    public int sum2(int x, int y){
        return x+y;
    }

    public int sum3(int x, int y, int z){
        return x+y+z;
    }

    ...
}

class Test{
    public static void main(String args[]){
        Operations op = new Operations();
        System.out.println("The sum of two numbers is : " +op.sum2(5,10));
        System.out.println("The sum of three numbers is : " +op.sum3(5,10,15));
    }
}
```

With method overloading:

```
class Operations{
    public int sum(int x, int y){
        return x+y;
    }

    public int sum(int x, int y, int z){
        return x+y+z;
    }

    ...
}

class Test{
    public static void main(String args[]){
        Operations op = new Operations();
        System.out.println("The sum of two numbers is : " +op.sum(5,10));
        System.out.println("The sum of three numbers is : " +op.sum(5,10,15));
    }
}
```



Calling the same method name but using different number of input parameters

Example: changing data types

```
class Operations{
    public int sum(int x, int y){
        return x+y;
    }

    public int sum(double x, double y){
        return x+y;
    }

    ...
}

class Test{
    public static void main(String args[]){
        Operations op = new Operations();
        System.out.println("The sum of two integer numbers is : " +op.sum(5,10));
        System.out.println("The sum of two floating point numbers is : " +op.sum(5.5,10.1));
    }
}
```

Example: changing parameter order

```
class Operations{
    public int sum(int x, double y){
        return x+y;
    }

    public int sum(double x, int y){
        return x+y;
    }

    ...
}

class Test{
    public static void main(String args[]){
        Operations op = new Operations();
        System.out.println("The sum of two numbers, int then double: " +op.sum(5,10.5));
        System.out.println("The sum of two numbers, double then int: " +op.sum(5.5,10));
    }
}
```

Example: overloading constructors

```java
class Rectangle{
    private int length;
    private int width;

    Rectangle(){
        this.length = 1;
        this.width = 1;
    }

    Rectangle(int x, int y){
        this.length = x;
        this.width = y;
    }

    public void rectArea(){
        int result;
        result = length * width;
        System.out.println("Area of rectangle is: " +result);
    }
}

class ConstructorDemo{
    public static void main(String args[]){
        Rectangle defaultRect = new Rectangle();
        defaultRect.rectArea();

        Rectangle parameterizedRect = new Rectangle(10,20);
        parameterizedRect.rectArea();
    }
}
```

Output:

```
Area of rectangle is: 1
Area of rectangle is: 200
```

Example: overloading constructor, allow copying one object into another

```
class Student{
    private int id;
    private String name;
    private double gpa;

    Student(int i, String n, double g){
        this.id = i;
        this.name = n;
        this.gpa = g;
    }

    Student(Student s){
        this.id = s.id;
        this.name = s.name;
        this.gpa = s.gpa;
    }

    void display(){
        System.out.println(id + " " + name + "(" + gpa + ")");
    }
}

class Test{
    public static void main(String args[]){
        Student s1 = new Student(101, "John Smith", 3.9);
        Student s2 = new Student(s1);
        s1.display();
        s2.display();
    }
}
```

Output:

```
101 John Smith(3.9)
101 John Smith(3.9)
```

Dynamic polymorphism (method overriding)

- Default implementation of the method is available to the child through inheritance
  - Child can redefine the method by the same name as the parent already implemented
    - Parent's method is called "overridden"
    - Child's method is called "overriding"
  - Child's method has to have the same parameters as the parent's method

Example: overriding method *sound()*

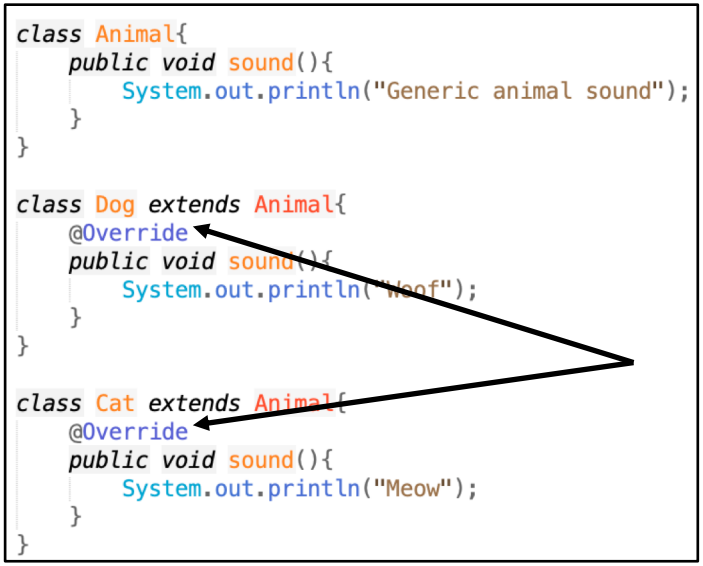
```
class Animal{
    public void sound(){
        System.out.println("Generic animal sound");
    }
}

class Dog extends Animal{
    public void sound(){
        System.out.println("Woof");
    }
}

class Cat extends Animal{
    public void sound(){
        System.out.println("Meow");
    }
}

class Test{
    public static void main(String args []){
        Dog barker = new Dog();
        barker.sound();

        Cat whiskers = new Cat();
        whiskers.sound();
    }
}
```

Normally we use *@Override* keyword (explained on the next slide)

Use of @Override in Java

- Used when overriding a method in subclass
- Considered a good practice
  - If any mistake is made in overriding there will be a compile error
  - Improves readability

In the previous example:



Static vs. dynamic polymorphism

Static polymorphism	**Dynamic polymorphism**
Happens at compile time	Happens at runtime
Overloading done in the same class	Overriding is done through inheritance
Parameters differ between methods with the same name	Parameters are the same as in the inherited method
Return data type can be different between methods with the same name	Return data type has to be the same in the inherited method
Main purpose: increase readability and convenience	Main purpose: give child class ability to define its own implementation without changing the parent's implementation
Private, static, and final method **can** be overloaded	Private, static, and final method **cannot** be overridden

Coupling in Java

- Loose coupling – degree to which the class depends on other classes
- Independent classes have separate behaviors
- In a system classes need to interact
- Loose coupling is important principle in OOD
  - Tight coupling == high dependency between classes
    - Two classes often change together
    - Classes ”know” a lot about each other
  - Loose coupling == classes are independent
    - Single responsibility
    - Separation of duties
- Classes are tightly coupled if
  - They interact through shared data
  - Method calls pass large amounts of shared data between classes

Loose coupling is recommended

- Loose coupling allows for reusability and extensibility
- Easier to swap/change code modules in loosely coupled design
- Loose coupling comes useful when the application changes or grows
  - E.g. requirement changes
- Over time the ease of maintenance of the application goes a long way!

Real life analogies of coupling

- Skin vs. shirt
  - Tight coupling
    - Imagine needing to change your skin?
  - Loose coupling
    - Like changing a shirt
- Car parts
  - In tightly coupled architecture you need to redesign the entire car if you just want to change/modify one part
  - In loosely coupled architecture you just replace that part and everything else works

How we achieve loose coupling

- Abstraction
  - E.g. interfaces
    - Allows changing the underlying implementations or introduce new ones without impacting the classes that implement interfaces

- Designated non-reusable class that is responsible for interactions

Example: shopping cart

Tightly coupled architecture:

```
public class CartEntry
{
    public float Price;
    public int Quantity;
}

public class CartContents
{
    public CartEntry[] items;
}

public class Order
{
    private CartContents cart;
    private float salesTax;

    public Order(CartContents cart, float salesTax)
    {
        this.cart = cart;
        this.salesTax = salesTax;
    }

    public float OrderTotal()
    {
        float cartTotal = 0;
        for (int i = 0; i < cart.items.Length; i++)
        {
            cartTotal += cart.items[i].Price * cart.items[i].Quantity;
        }
        cartTotal += cartTotal*salesTax;
        return cartTotal;
    }
}
```

Order class needs to know the internal architecture of CartContents class

Order class does not know the innerworkings of CartContents class

Loosely coupled architecture:

```
public class CartEntry
{
    public float Price;
    public int Quantity;

    public float GetLineItemTotal()
    {
        return Price * Quantity;
    }
}

public class CartContents
{
    public CartEntry[] items;

    public float GetCartItemsTotal()
    {
        float cartTotal = 0;
        foreach (CartEntry item in items)
        {
            cartTotal += item.GetLineItemTotal();
        }
        return cartTotal;
    }
}

public class Order
{
    private CartContents cart;
    private float salesTax;

    public Order(CartContents cart, float salesTax)
    {
        this.cart = cart;
        this.salesTax = salesTax;
    }

    public float OrderTotal()
    {
        return cart.GetCartItemsTotal() * (1.0f + salesTax);
    }
}
```

https://stackoverflow.com

Example: shape area

Introducing abstraction helps make sure ShapeSorter class does not need to know about the innerworkings of the other classes

Tightly coupled architecture:

```
public class Rectangle {
    public void computeArea() {
        System.out.println("Computing rectanle area");
    }
}

public class Circle {
    public void computeArea() {
        System.out.println("Computing circle area");
    }
}

public class ShapeSorter {
    public void computeShapeArea(Rectangle r) {
        r.computeArea();
    }

    public void computeShapeArea(Circle c) {
        c.computeArea();
    }
}
```

Loosely coupled architecture:

```
public interface Shape {
    public void computeArea();
}

public class Rectangle implements Shape {
    public void computeArea() {
        System.out.println("Computing rectanle area");
    }
}

public class Circle implements Shape {
    public void computeArea() {
        System.out.println("Computing circle area");
    }
}

public class ShapeSorter {
    public void computeShapeArea(Shape myShape) {
        myShape.computeArea();
    }
}
```

OOD principles

- Cohesion
- Completeness
- Clarity
- Convenience
- Consistency

Cohesion principle

- Class is cohesive if attributes and methods of a class cooperate to achieve a common purpose
- Closely related to encapsulation principle
- Strong cohesion is important in good OOD
- A class with varying or conflicting functions is considered to be poorly designed
- Too much functionality decreases cohesion
- Cohesion makes classes easier to manage
  - modular system design

Completeness principle

- Class contains all operations important to the abstraction it represents
- Consider future uses of the class

Clarity principle

- Class interface should be clear to other programmers
  - Avoid ambiguity
  - Clear method names that tell users what operations they perform
  - Put yourself into the shoes of your class users
    - They have no insight into your class black box (encapsulation)

Convenience principle

- Strive for ease of use for your class
- Know your target audience
- How easy to achieve different tasks that your class does?

Consistency principle

- Consistency among class methods/method calls
- Consistency in naming convention
- Consistency among class operations/behavior
- Clarity is important! Consistency provides clarity

Information hiding principle

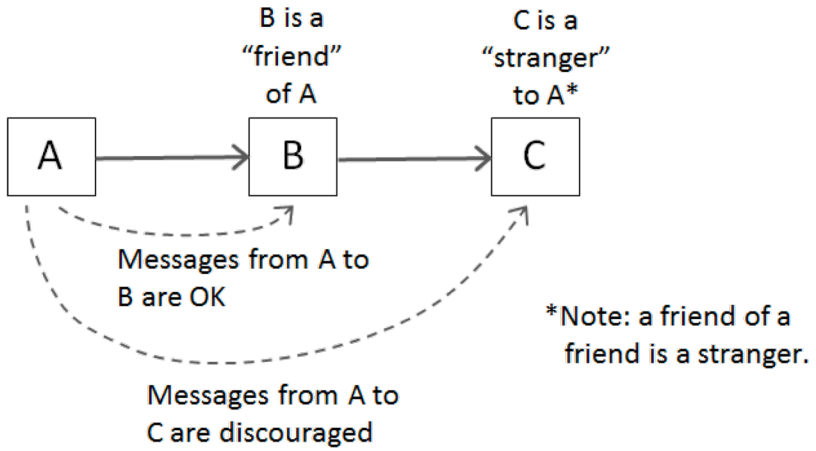
- General computer science principle
- Segregation of design decisions that are more likely to change
- In OOD world, dictates that certain parts of a class should not be accessible to users
- Directly related to encapsulation

Law of Demeter (LoD)

- Proposed by Ian Holland at Northeastern University in 1987
- Can be viewed as ”principle of least knowledge” guidelines
- Summarized as (Wikipedia):
  - Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
  - Each unit should only talk to its friends; don't talk to strangers.
  - Only talk to your immediate friends.
- Follows from the information hiding principle
- Objects assume and know as little about the internal structure or components of other objects as possible

Law of Demeter (LoD) cont'd

- Class A should not use class B to directly access functionality of class C
  - Instead use a message propagation A -> B -> C
  - Should not see things like *a.getX().getY()*
- Any method of the class should only call methods:
  - Of itself
  - Of the objects that are fields/data of the class
  - Of the objects that are passed into this method
  - Of the object created inside this method



Carlos Caballero; https://dev.to/carlillo/demeters-law-dont-talk-to-strangers-10ep

To recap

- OOP concepts
  - Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism
- OOD principles of class design
  - Cohesion
  - Completeness
  - Clarity
  - Convenience
  - Consistency
- Important points to know
  - Classes and interfaces represent abstraction in OOP
  - Interfaces allow bypassing of restrictions to multiple inheritance
  - Java packages are a useful way of encapsulation
  - Loose coupling is important in OOP

Concluding remarks

- Good object oriented design impacts software quality
- Best way to learn OOP is to practice
- Game design/programming is a naturally good fit for OOD
  - Entities in game world can be represented with classes/objects
  - Message passing between entities resembles message passing between objects