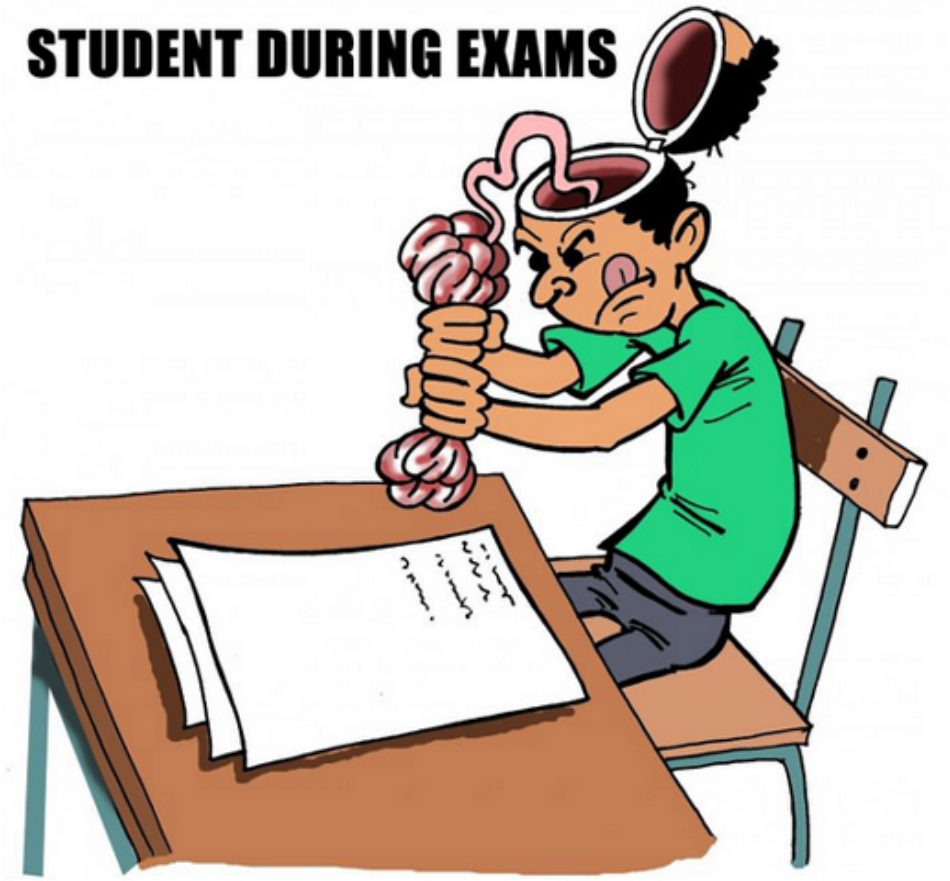
CSE201: Monsoon 2020, Section-A
Advanced Programming

Lecture 14: Mid Semester Review

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

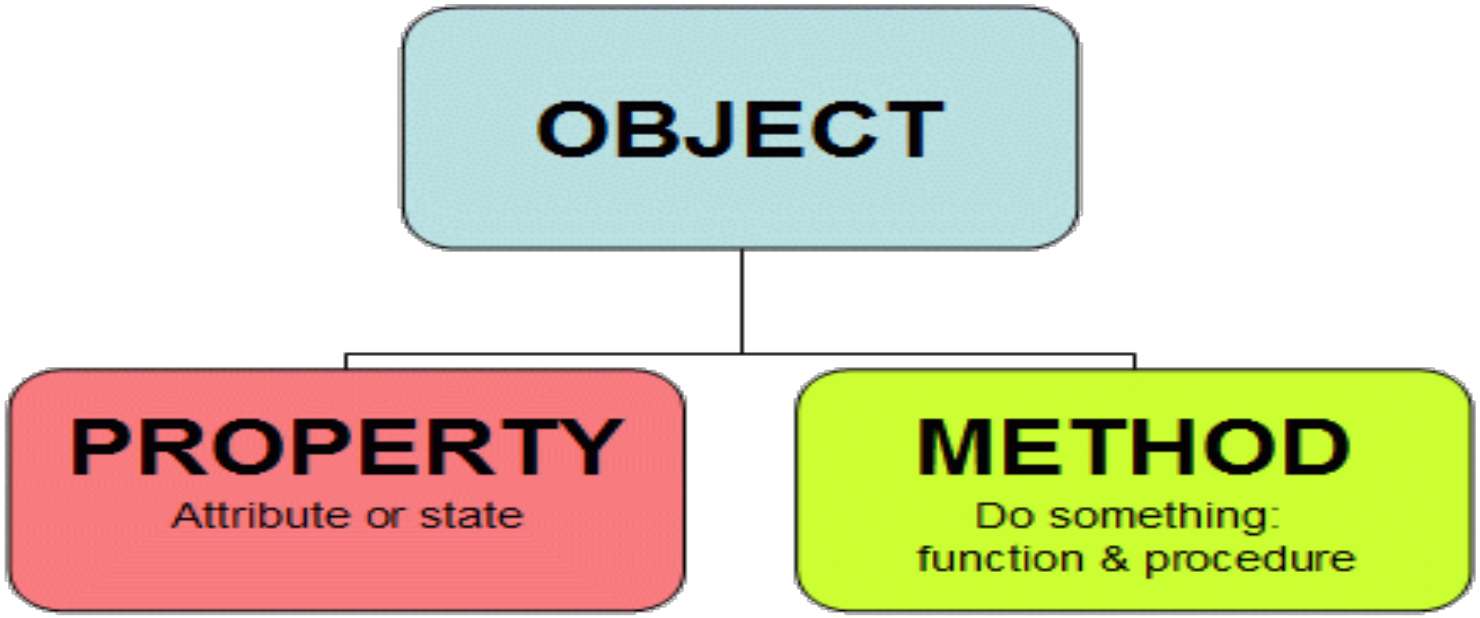
.
We Tried Our Best
Can't Say Anything Right Now!



This lecture is to help you in avoiding situations like this...

OOP: Classes and Objects

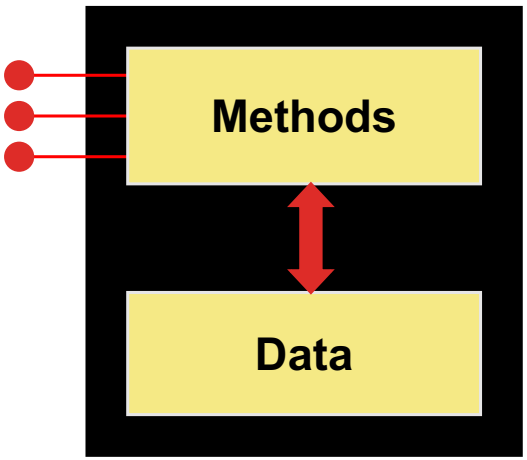
What is OOP?



It is a programming paradigm based on the concept of "objects", which may contain data in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods
(Wikipedia)

Encapsulation

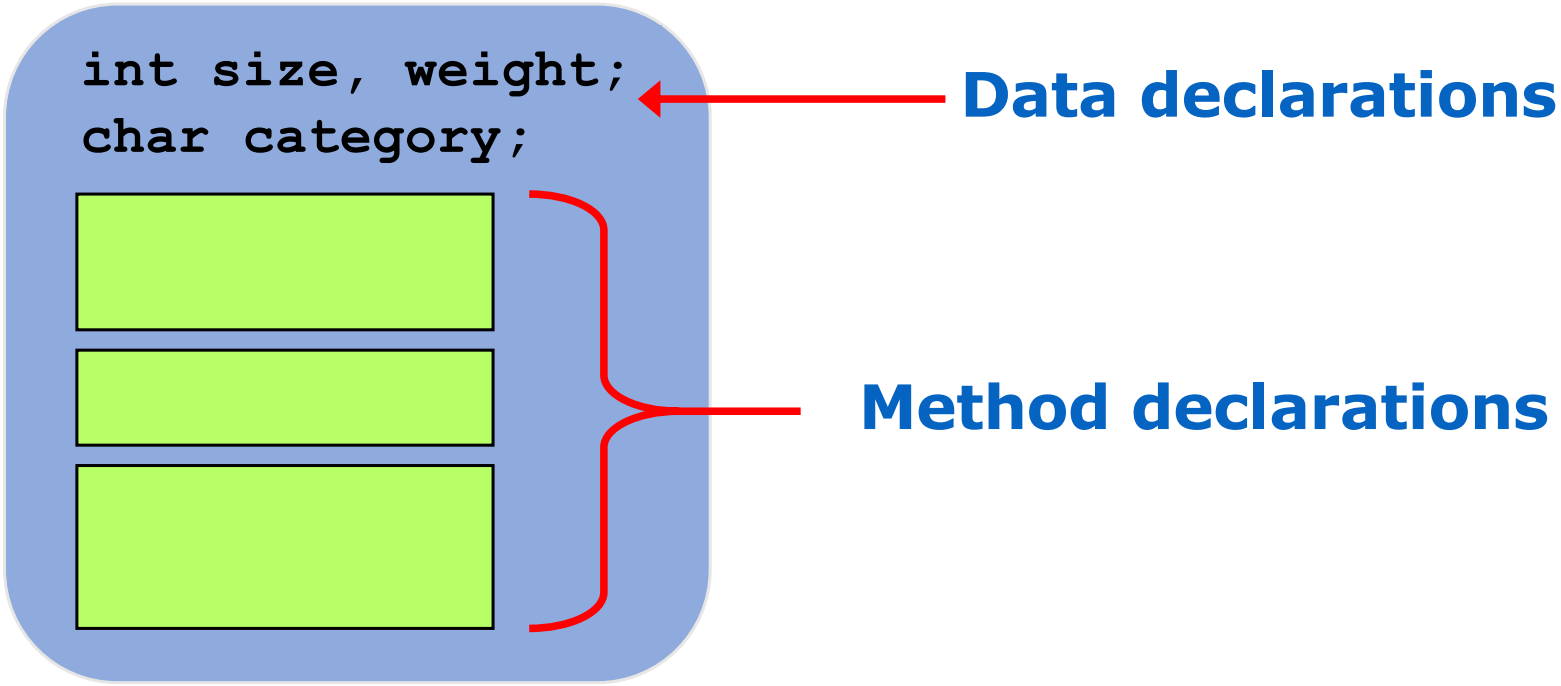
- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods of the object, which manages the instance data



Abstraction lets you focus on what the object does instead of how it does, while encapsulation means hiding the internal details of how an object works

Classes

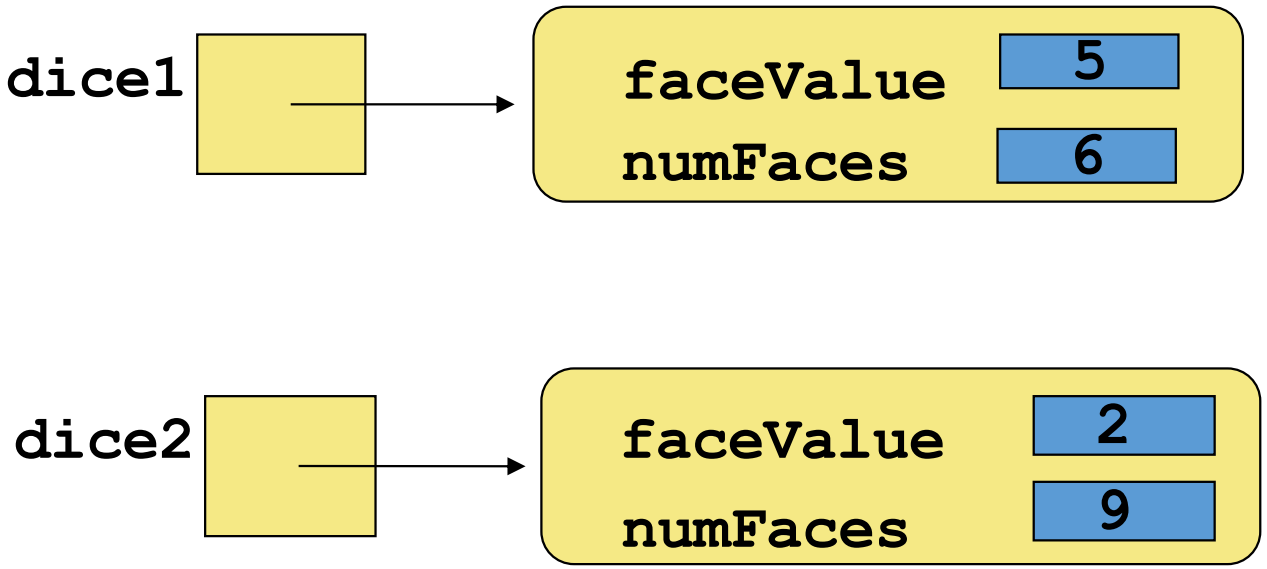
- A class can contain data declarations and method declarations



© 2004 Pearson Addison-Wesley. All rights reserved

Object Instances

- We can depict the two objects of `Dice` class as follows:



Each object maintains its own `faceValue` and `numFaces` variable, and thus its own state

Identifying Classes and Methods

For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click OK button. The email account will first validate the customer credentials and then grant access to his email account.

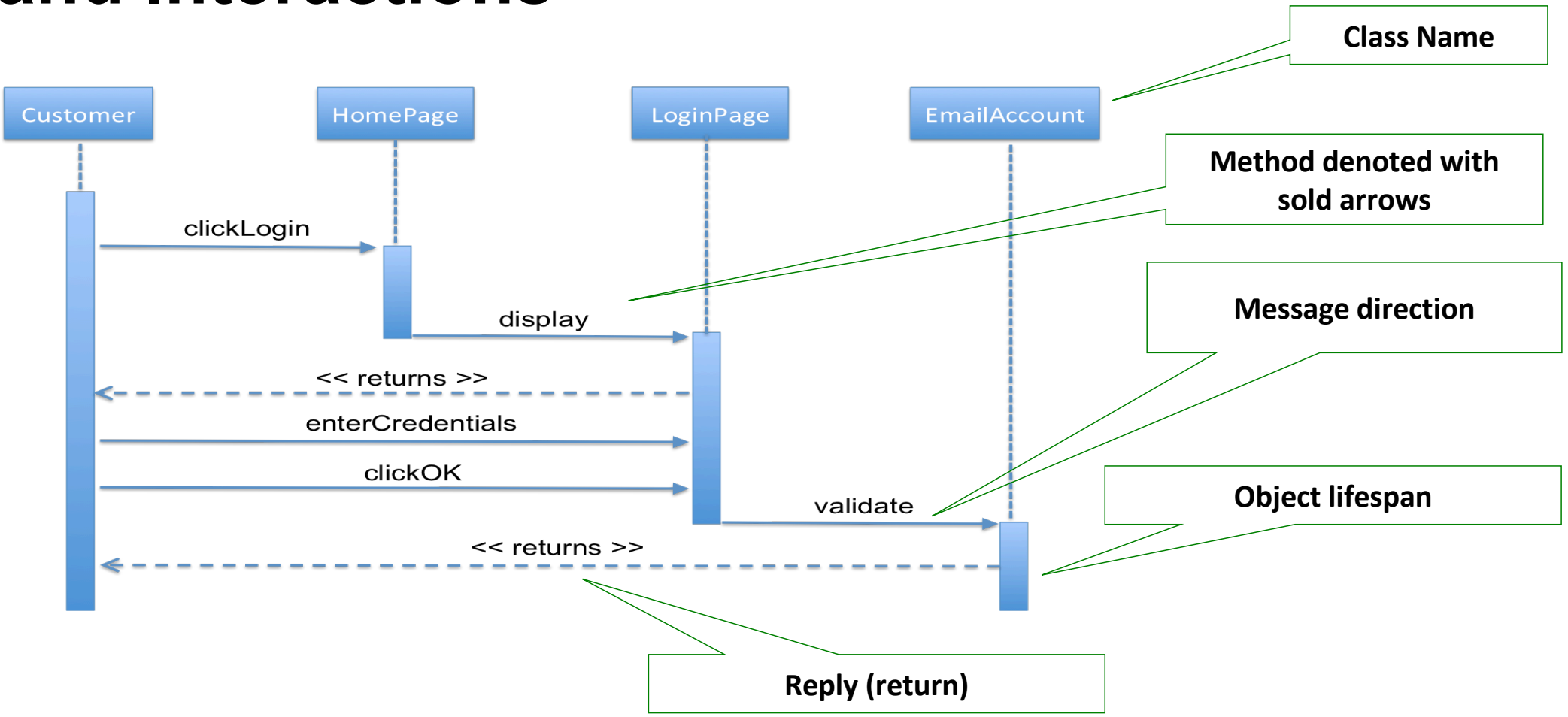
Classes
Customer
HomePage
LoginPage
EmailAccount

Methods
clickLogin
display
enterCredentials
clickOK
validate

- Classes
  - Class represents a group of objects with similar behaviors
  - Look for nouns
- Methods
  - Verbs

© Vivek Kumar

Sequence Diagram: Tracing Object Methods and Interactions



© Vivek Kumar

Class Relationships

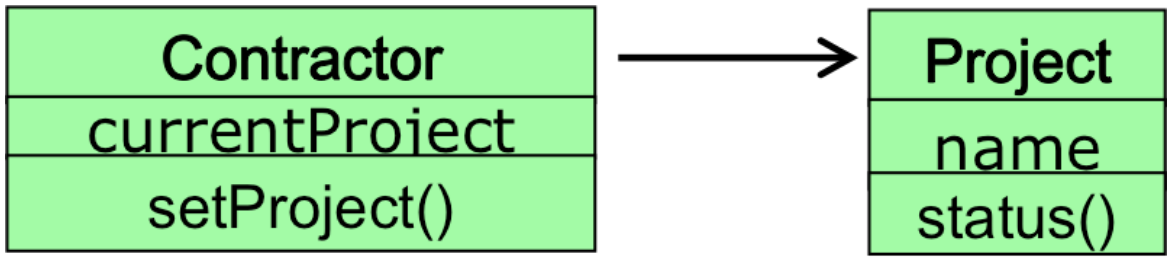
Class Relationships

- When writing a program, need to keep in mind “big picture”—how are different classes related to each other?
- Most common class relationships
  - Association
  - Composition
  - Dependency
  - Inheritance

© Vivek Kumar

Association Relationship

- Class A and class B are **associated** if A "knows about" B, but A does not **contains (instantiate)** object of B
  - But this is not symmetrical! B need not know about A
- **Class A holds a class level reference to class B**
- Lifetime?
  - Objects of class A and B have their own lifetime, i.e., they can exist without each other

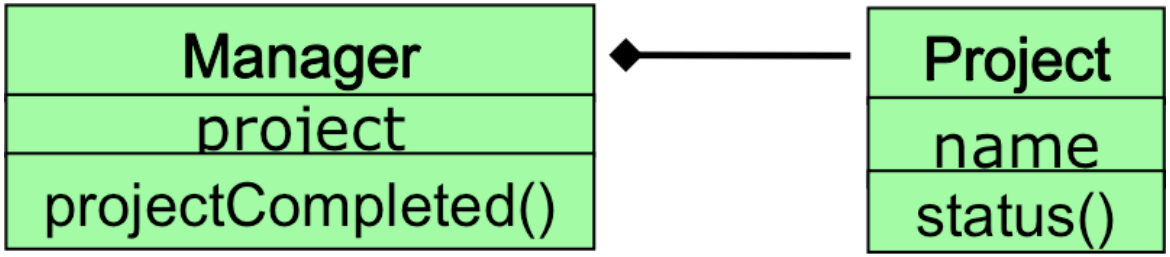



```
class Project {
   private String name;
   public boolean status() { ... }
   .....
}
// Contractor's project keep changing
class Contractor {
   private Project currentProject;
   public Contractor(Project proj) {
     this.currentProject = proj;
   }
   public void setProject(Project proj){
     this.currentProject = proj;
   }
}
```

© Vivek Kumar

Composition Relationship

- Class A **contains** object of class B
  - A **instantiate** B
  - But this is not symmetrical! B need not **contain/know-about** A
- Lifetime?
  - The death relationship
  - Garbage collection of A means B also gets garbage collected

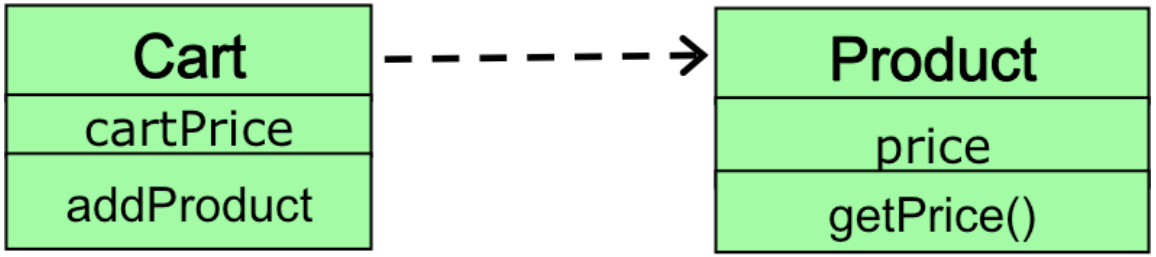



```
class Project {
   private String name;
   public boolean status() { ... }
   .....
}
// Contractor has a fixed project
class Contractor {
   private Project project;
   public Contractor() {
     this.project = new Project("ABC");
   }
   public boolean projectCompleted() {
     return project.status();
   }
}
```

© Vivek Kumar

Dependency Relationship

- Neither class A or class B **contains** or **know-about** each other
- Class A **depends** on class B if A cannot carry out its work without B
  - Need not be symmetrical! B doesn't depends on A
- Created when class A receives a reference to another class B as part of a particular operation or method



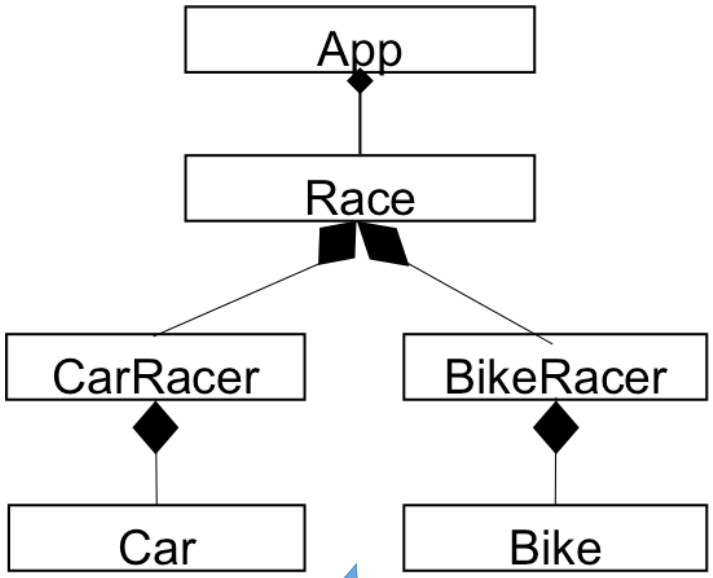

```
class Product {
   private double price;
   .....
   public double getPrice() { ..... }
}

class Cart {
   private double cartPrice;
   public void addProduct(Product p) {
     cartPrice += p.getPrice();
   }
}
```

© Vivek Kumar

Interfaces and Polymorphism

Motivation

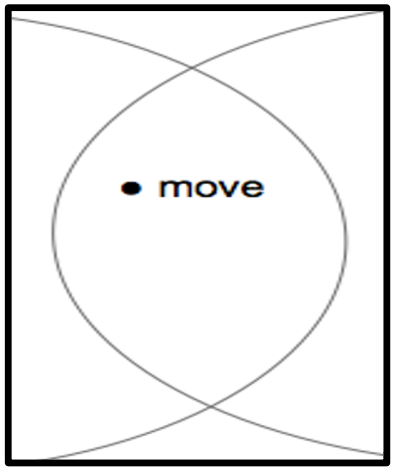


Do we need two different Racer classes??

```
public class Racer {

    public Racer() {
        //constructor
    }
    public void useCar(Car myCar){//code elided}
    public void useBike(Bike myBike){//code elided}
    public void useHoverboard(Hoverboard myHb){//code elided}
    public void useHorse(Horse myHorse){//code elided}
    public void useScooter(Scooter myScooter){//code elided}
    public void useMotorcycle(Motorcycle myMc) {//code elided}
    public void usePogoStick(PogoStick myPogo){//code elided}
    // And more…
}
```

How about one Racer class with different methods?



Any similarity?

Interfaces

Declaring an Interface

```
public interface Transporter {

    public void move();

}
```

Implementing an Interface

```
public class Car implements
Transporter {
    public Car() {
        //code elided
    }
    public void drive(){
        //code elided
    }
    @Override
    public void move(){
        this.drive();
        this.brake();
        this.drive();
    }
    //more methods elided
}
```

- Group similar capabilities/function of different classes together
- Interfaces can only declare methods - not define them
- Interfaces are contracts that classes agree to
- If classes choose to implement given interface, it must define all methods declared in interface
  - if classes don't implement one of interface's methods, the compiler raises error

`@Override` is an annotation – a signal to the compiler (and to anyone reading your code)

Interface and Polymorphism

```
public class App {
    public App() {
        Race r = new Race();
        r.startRace();
    }
}
```

```
public class Race {
    private Racer _dan, _sophia;

    public Race(){
        _dan = new Racer();
        _sophia = new Racer();
    }
    public void startRace() {
        _dan.useTransportation(new Car());
        _sophia.useTransportation(new Bike());
    }
}
```

```
public interface Transporter {
    public void move();
}
```

```
public class Racer {
   public Racer() {}

   public void useTransportation(Transporter transport){
       transport.move();
   }
}
```

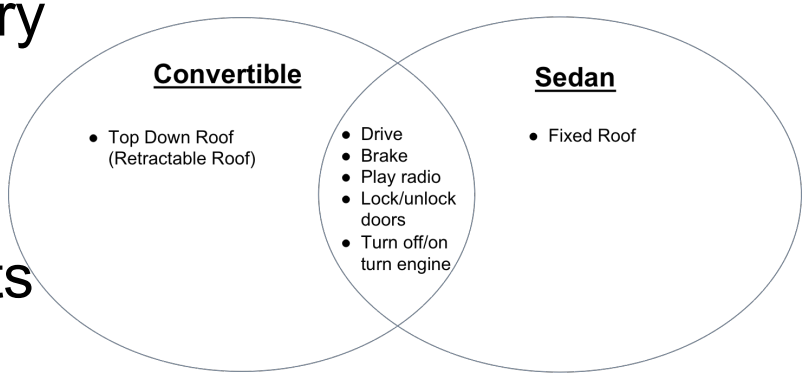
```
public class Car implements Transporter {
   public Car() {}
   public void drive() {
      //code elided
   }
   public void move() {
       this.drive();
   }
}
```

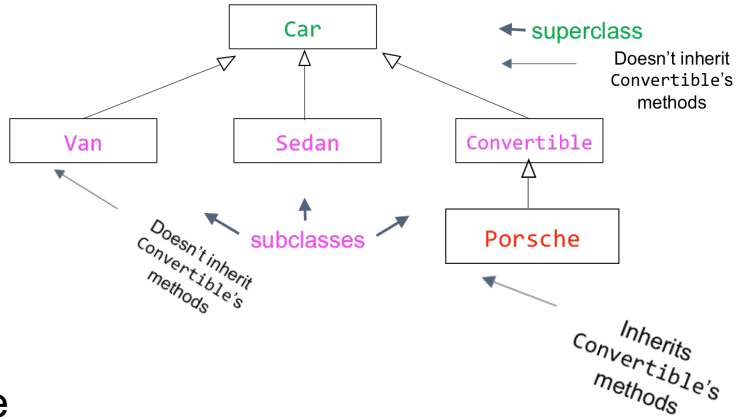
```
public class Bike implements Transporter {
   public Bike() {}
   public void pedal() {
       //code elided
   }
   public void move() {
       this.pedal();
   }
}
```

Inheritance and Polymorphism

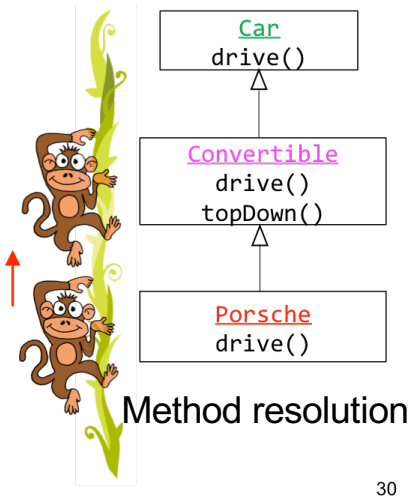
Inheritance

- In OOP, inheritance is a way of modeling very similar classes
- **Superclass/parent/base**: A class that is inherited from
- **Subclass/child/derived**: A class that inherits from another
- A subclass inherits all of its parent's **public** and **protected** capabilities
- Inheritance and Interfaces both legislate class's behavior, although in very different ways
  - Interfaces allow the compiler to enforce method implementation
    - An implementing class will have all capabilities outlined in an interface
  - Inheritance assures the compiler that all **subclasses** of a **superclass** will have the superclass's public/protected capabilities without having to respecify code – methods are inherited





Inheritance and Polymorphism



Method resolution

30

```
public class Car {

    private Engine _engine;
    //other variables elided

    public Car() {
        _engine = new Engine();
    }
    public void drive() {
        this.goFortyMPH();
    }
    public void goFortyMPH() {
        //code elided
    }
    protected void cleanEngine()
{ ... }
}
```

```
public class Convertible extends Car {

    public Convertible(){

    }

    public void putTopDown(){
        //code elided
    }
}
```

18

```
public class Convertible extends Car {
    //constructor elided
    public void cleanCar() {
        _engine.steamClean();
    }
}
```

```
public class Sedan extends Car {

    public Sedan () {
        //code elided
    }

    @Override
    public void drive(){
        this.turnOnEngine();
        super.drive(); // super == parent
                       class

        this.addPinToMap();
        super.drive();
        super.drive();
        this.addPinToMap();
    }

}
```

28

```
public class Racer {

    //previous code elided

    public void useTransportation(Car myCar) {
        myCar.drive();
    }

}
```

- Adding new methods
- Accessing superclass fields/methods
- Overriding superclass methods
- Polymorphism
- Method resolution

Abstract Class

- We declare a method `abstract` in a superclass when the subclasses can't really re-use any implementation the superclass might provide
- Any class having an `abstract` method is an abstract class and is denoted using `abstract` keyword
- Abstract classes cannot be instantiated but its constructor must still be invoked via `super()` by a subclass
- Subclass at any level in inheritance hierarchy can make `abstract` method concrete by providing implementation
- Abstract class v/s interfaces
  - Can define instance variables unlike interfaces
  - Can define a mix of concrete and `abstract` methods, unlike interfaces where you cannot have any concrete method
  - You can only inherit from one class whereas you can implement multiple interfaces

Abstract Class and Methods

```
public class Convertible extends Car{
    @Override
    public void loadPassengers(){
        Passenger p1 = new Passenger();
        p1.sit();
    }
}
```

```
public class Sedan extends Car{
    @Override
    public void loadPassengers(){
        Passenger p1 = new Passenger();
        p1.sit();
        Passenger p2 = new Passenger();
        p2.sit();
    }
}
```

```
public class Van extends Car{
    @Override
    public void loadPassengers(){
        Passenger p1 = new Passenger();
        p1.sit();
        Passenger p2 = new Passenger();
        p2.sit();
        Passenger p3 = new Passenger();
        p3.sit();
    }
}
```

- All concrete subclasses of `Car` override by providing a concrete implementation for `Car`'s abstract `loadPassengers()` method
- As usual, method signature must match the one that `Car` declared

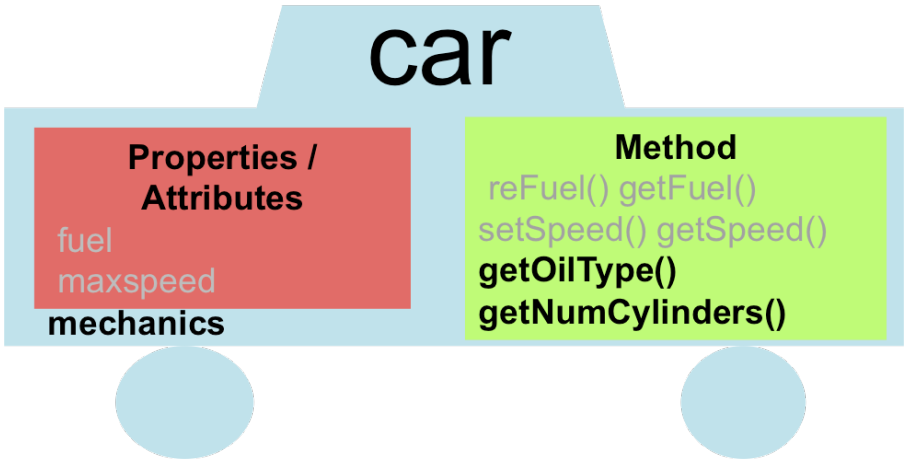
Andries van Dam © 2016 9/22/16

Making a Class Immutable

1. Don't provide any methods that modify the object's state.
2. Make all fields `private`. (ensure encapsulation)
3. Make all fields `final`.
4. Ensure exclusive access to any mutable object fields.
   - Don't let a client get a reference to a field that is a mutable object (don't allow any mutable representation exposure.)
5. Ensure that the class cannot be *extended*.

© Vivek Kumar

Immutable Class




```
public class final Mechanics {
    private final String oilType;
    private final int numCylinders;

    public Mechanics(String oil, int cylinders)
    public String getOilType();
    public int getNumCylinders();
}
```

Mechanics cannot be extended as it is declared as final

```
public class ModifiedMechanics extends Mechanics {

  ......
  @Override
  public String getOilType(){
      return "Rocket Fuel";
  }
  @Override
  public int getNumCylinders(){return 18;}//Bugatti
}
```


© Vivek Kumar

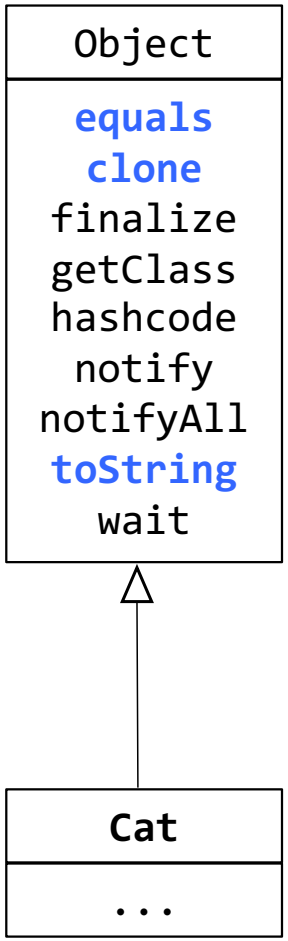
Object Comparison and Copying

The Class Object

- The class `Object` forms the root of the overall inheritance tree of all Java classes.
  - Every class is implicitly a subclass of `Object`
  - `No need to explicitly say “extends Object”`

- The `Object` class defines several methods that become part of every class you write. For example:

  - `public String toString()`
    Returns a text representation of the object, usually so that it can be printed.



© Vivek Kumar

The `equals` Method in Object Class

```
1. public class Point {
2.    private int x, y;
3.    public Point(int _x, int _y) { ... }
4.    @Override
5.     public boolean equals(Object o1) {
6.        if(o1 != null && getClass() == o1.getClass()) {
7.          Point o = (Point) o1; //type casting
8.          return (x==o.x && y==o.y);
9.        }
10.       else {
11.         return false;
12.       }
13.    }
14.  }
15.  // subclass of Point
16.  class Point3D extends Point {
17.     private int z;
18.     public Point3D(int _z) { ... }
19.     @Override
20.     public boolean equals(Object o1) {
8.        if(o1 != null && getClass() == o1.getClass()) {
9.          Point3D o = (Point3D) o1; //type casting
8.          return (super.equals(o1) && z==o.z);
9.        }
10.       else {
11.         return false;
12.       }
13.    }
14.  }
```

- `getClass` returns information about the type of an object
  - Stricter than `instanceof`; subclasses return different results
- `getClass` should be used when implementing equals
  - Instead of `instanceof` to check for same type, use `getClass`
  - This will eliminate subclasses from being considered for equality
  - Caution: Must check for `null` before calling `getClass`

© Vivek Kumar

Comparable Example

```
public class Rectangle implements Comparable<Rectangle> {
    private int sideA, sideB, area;
    public Rectangle (int _a, int _b) { ... }

    @Override
    public int compareTo(Rectangle o) {
      if(area == o.area) return 0;
      else if(area < o.area) return -1;
      else return 1;
    }
}
```

- In this Rectangle class, the compareTo method compares the Rectangle objects as per their area
- You can choose your own comparison algorithm!

© Vivek Kumar

Comparator Example

```
public class RectangleAreaComparator
                        implements Comparator<Rectangle> {
    @Override
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}
```

```
public class RectangleSidesComparator
                        implements Comparator<Rectangle> {
    @Override
    public int compare(Rectangle r1, Rectangle r2) {
        if (r1.getSideA() != r2.getSideA()) {
            return r1.getSideA() - r2.getSideA();
        } else {
            return r1.getSideB() - r2.getSideB();
        }
    }
}
```

© Vivek Kumar

- Using Comparators, two objects could be compared in different possible ways
- For creating different comparison, implement different objects of Comparator type

Object Clonning

```
public class BankAccount implements Cloneable {
   private String name;
   private List<String> transactions;
   ...
   public BankAccount clone() {
      try {                  // deep copy
         BankAccount copy = (BankAccount) super.clone();
         copy.transactions = new ArrayList<String>(transactions);
         return copy;
      } catch (CloneNotSupportedException e) {
         return null;   // won't ever happen
      }
   }
}
```

- Copying the list of transactions (and any other modifiable reference fields) produces a deep copy that is independent of the original.

Generics and Collection Framwork

Generic Programming

- Our generic cup can hold different types of liquid
- In the notation `Cup<T>`:
  - T = Coffee
  - T = Tea
  - T = Milk
  - T = Soup
  - ……

Cup == Generic Container

© Vivek Kumar

Generic Programming

```
public class Pair <T1, T2> {
    private T1 key;
    private T2 value;
    public Pair(T1 _k, T2 _v) {
       key = _k; value = _v;
    }
    public T1 getKey() {  return key; }
    public T2 getValue() { return value; }
}
```

```
public class Main {
    public static void main(Sting args[]) {
        MyGenericList<Pair<String, Integer>> db =
                    new MyGenericList<Pair<String, Integer>>();
        db.add(new Pair<String, Integer>("John", 2343));
        db.add(new Pair<String, Integer>("Susane", 8908));
        ...
    }
}
```

- This is usage of a generic class with multiple fields
- Restrictions
  - Type parameters cannot be instantiated with primitive types
  - Instantiating type variables is not allowed
  - Generic array creation is not allowed
  - Type variables are not valid as static field of a generic class
  - Generic does not supports sub typing

© Vivek Kumar

Exception Handling

Basic Exception Handling

- Exception handling
  - To catch runtime errors
  - **try / catch / finally** block to exception handling
  - **try/catch** blocks could be nested
  - Single **try** could have multiple **catch** blocks
  - Methods can **throw** exceptions

```
public class Andy {
    public void getWater() {
        try {
            _water = _wendy.getADrink();
            int volume = _water.getVolume();
        }
        catch(NullPointerException e) {
            this.fire(_wendy);

            try {
                _water = johny.getADrink();
                int volume = _water.getVolume();
            }
            catch(NullPointerException e) {
                this.fire(johny);
            }
        }
    }
}
```

```
public class Andy {
    .....
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NullPointerException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() {
        try {
            _water = _wendy.getADrink();
            int volume = _water.getVolume();
        }
        catch(NullPointerException e) {
            this.fire(_wendy);
            System.out.println("Wendy is fired!");
            throw new NullPointerException("NO Water");
        }
    }
}
```

© Vivek Kumar

Advanced Exception Handling

- `Exceptions` are classes that extends `Throwable`
  - **Checked exceptions**
    - Those that **must** be handled somehow (e.g., `IOException`)
  - **Unchecked exceptions**
    - Those whose handling isn't mandatory (e.g., `RuntimeExceptions`)
      - You should **not** attempt to handle exceptions from subclass of `Error`
- Golden rules for using "`throws`" in method declaration
  - **Any method** that calls another method capable of generating **checked exceptions,** then the caller method must either try/catch the exception or declare the list of those checked exceptions using "`throws`" statement
  - In inheritance, if an overridden method in child class `throws` **checked exceptions**, then declaration of this method in parent should also declare those **checked exceptions** using `throws`

© Vivek Kumar

Defining Your Own Exception

```
public class NoWaterException extends Exception {
    public NoWaterException(String message) {
        super(message);
    }
}
public class Andy {
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NoWaterException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() throws NoWaterException {
        _water = _wendy.getADrink();
        if(_water == null) {
            this.fire(_wendy);
            throw new NoWaterException("NO Water");
        }
    }
}
```

© Vivek Kumar

- Useful for responding to special cases, not covered by pre-defined exceptions
- Every method that throws `Exceptions` that are not subclasses of `RuntimeException` must declare what exceptions it throws in method declaration (see golden rules in previous slide)
  - `getWater()` is throwing the exception, hence it must declare that using the "`throws`" on method declaration

All the best for your exam !!