CSE201: Monsoon 2024
Advanced Programming

End Semester Review

Dr. Arun Balaji Buduru

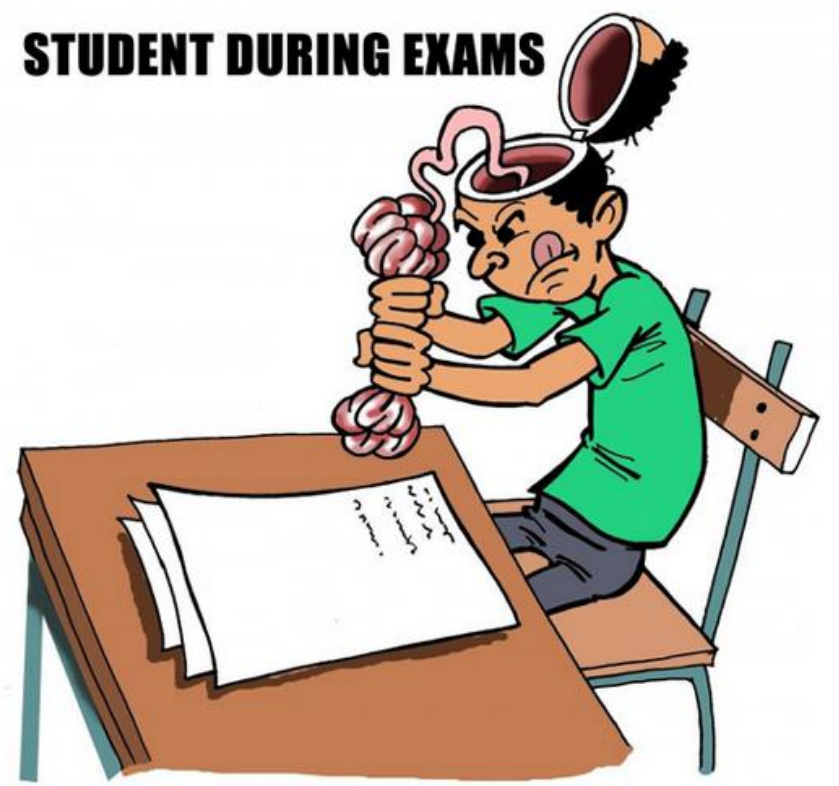
Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIIT-Delhi, India

.

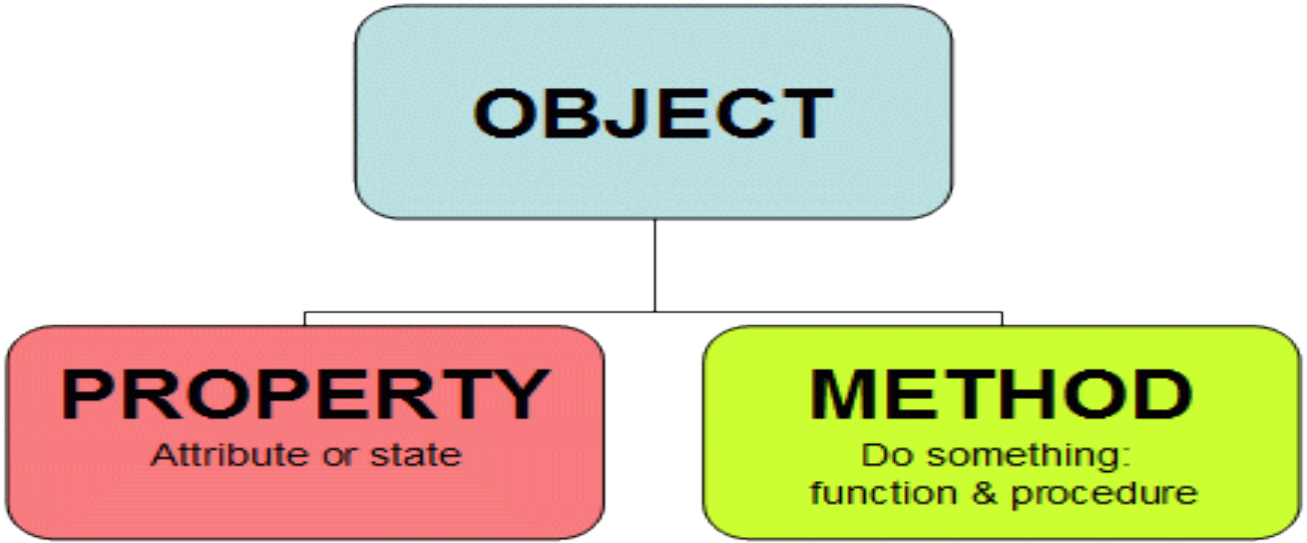
We Tried Our Best
Can't Say Anything Right Now!



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

OOP: Classes and Objects

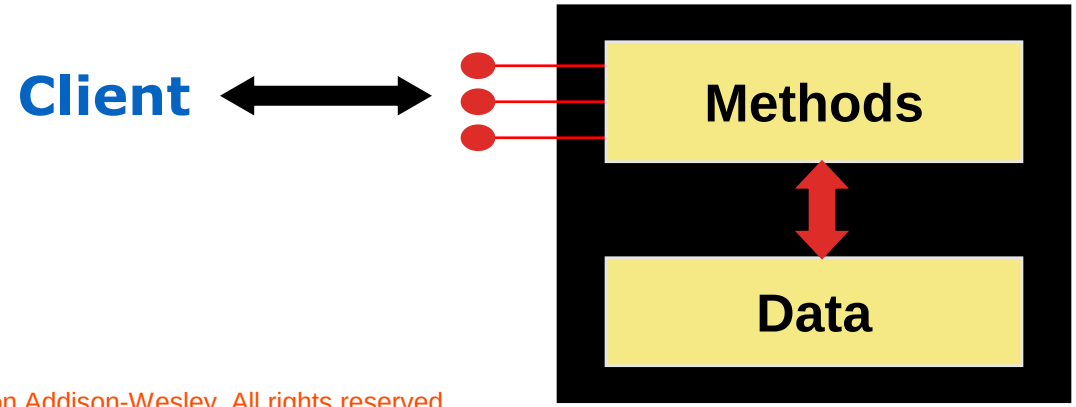
What is OOP?



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)

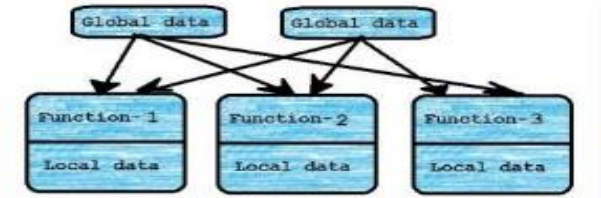
Encapsulation

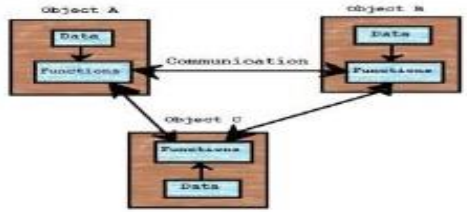
- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods of the object, which manages the instance data



Procedural v/s OOP

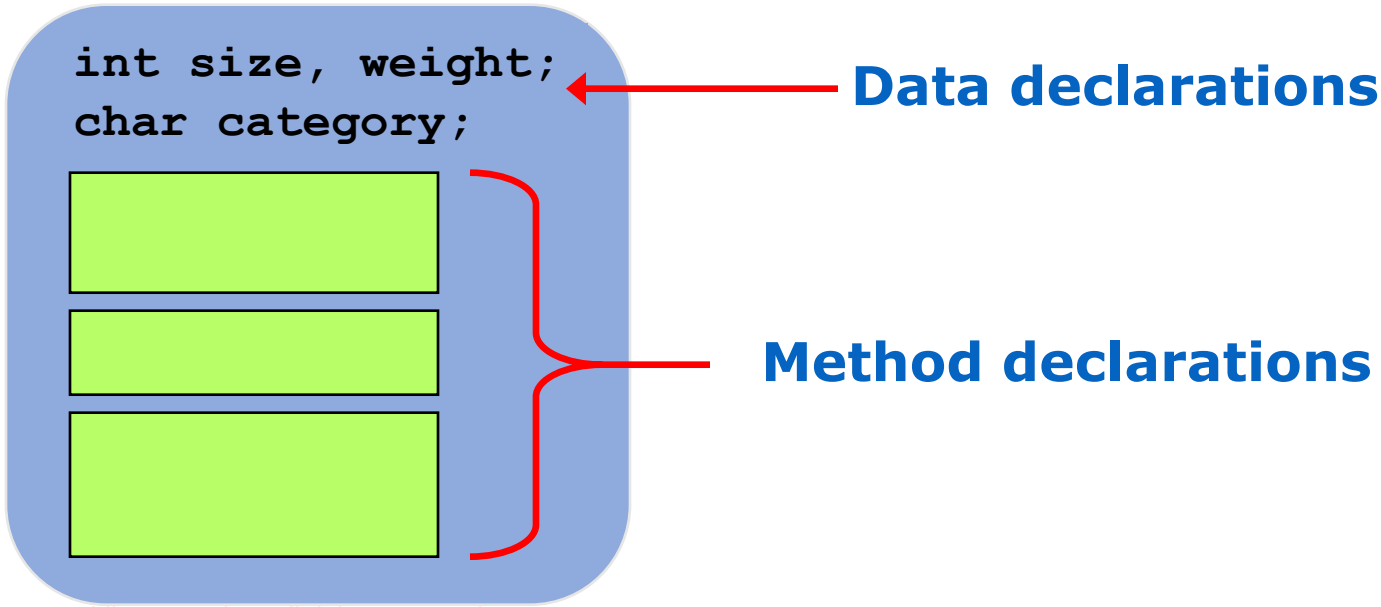
PROCEDURAL PROGRAMMING	OBJECT ORIENTED PROGRAMMING
11.Relationship of data and function in procedural	11.Relationship of data and function in OOP.





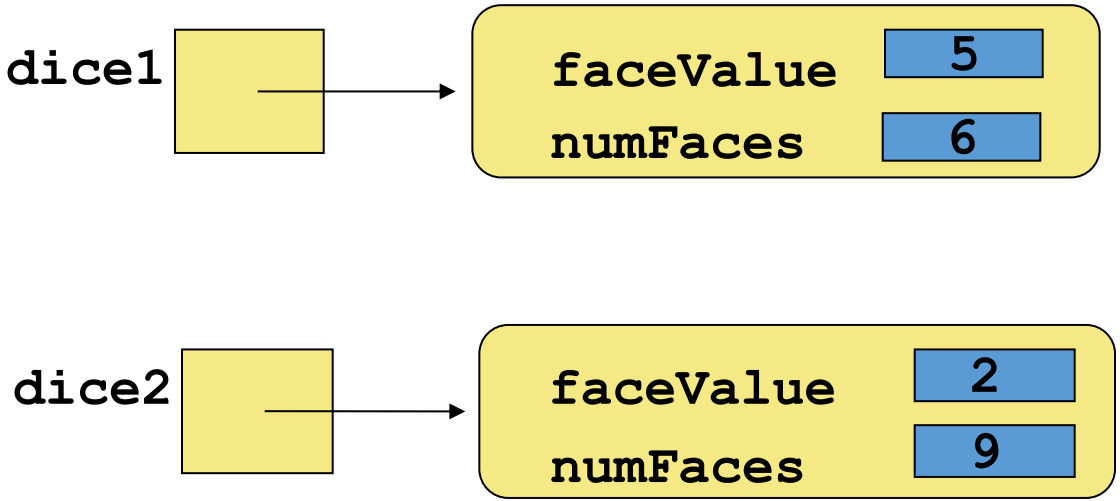
Classes

- A class can contain data declarations and method declarations



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

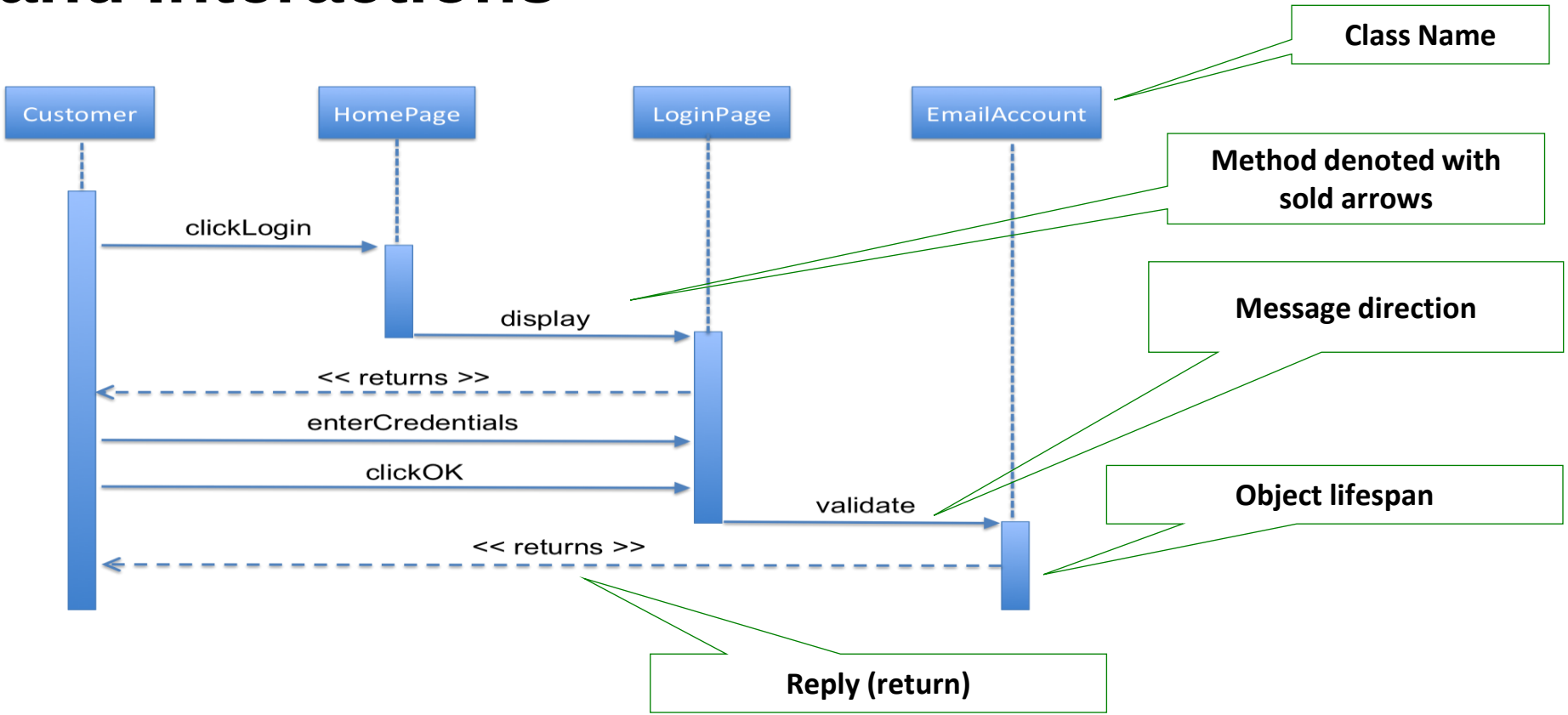
For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click OK button. The email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

Sequence Diagram: Tracing Object Methods and Interactions



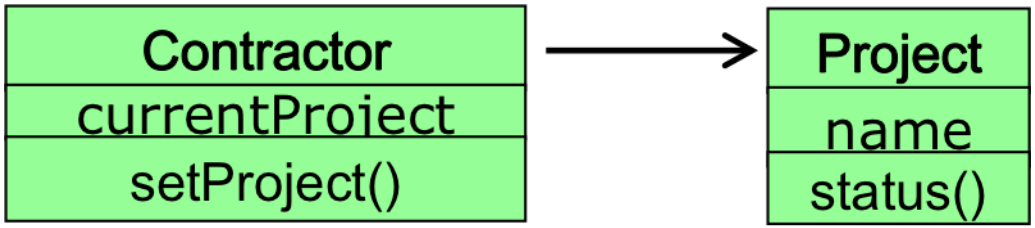
Class Relationships

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

Association Relationship

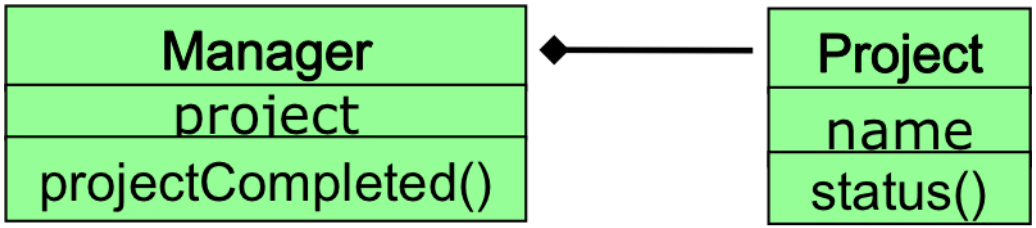
- Class A and class B are **associated** if A “knows about” B, but A does not **contains (instantiate)** object of B
  - But this is not symmetrical! B need not know about A
- **Class A holds a class level reference to class B**
- Lifetime?
  - Objects of class A and B have their own lifetime, i.e., 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;
   }
}
```

Composition Relationship

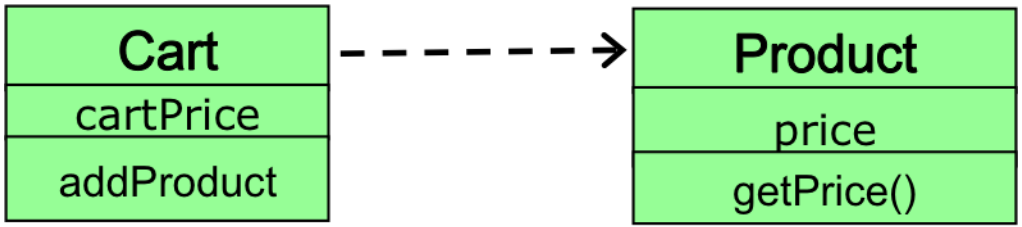
- Class A **contains** object of class B
  - A **instantiate** B
  - But this is not symmetrical! B need not **contain/know-about** A
- Lifetime?
  - The death relationship
  - Garbage collection of A means B also gets garbage collected




```
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();
   }
}
```

Dependency Relationship

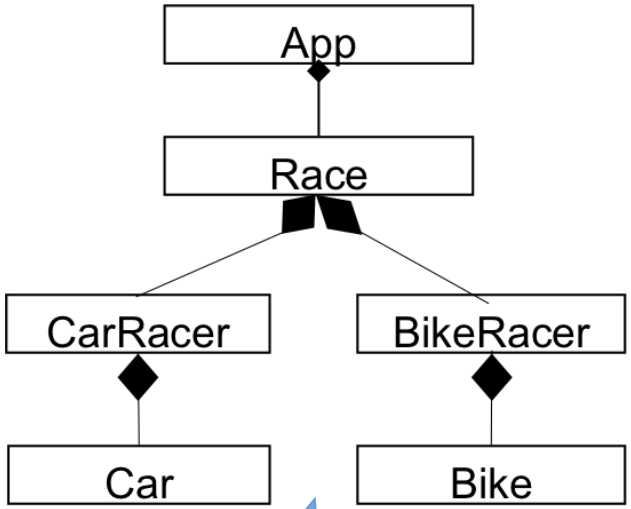
- Neither class A or class B **contains** or **know-about** each other
- Class A **depends** on class B if A cannot carry out its work without B
  - Need not be symmetrical! B doesn't depends on A
- Created when class A receives a reference to another class B as part of a particular operation or method




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

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

Interfaces and Polymorphism

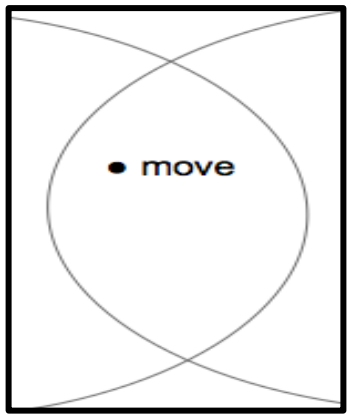
Motivation



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();
   }
}
```

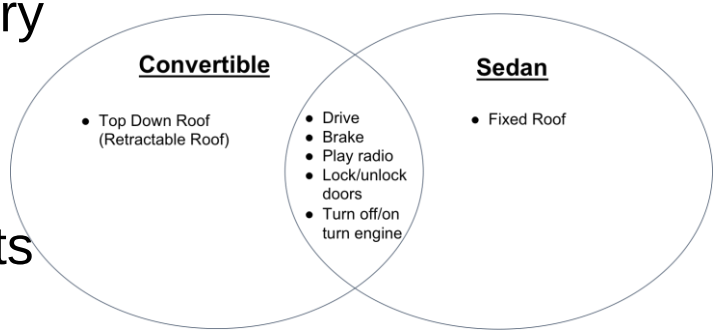
```
public class Car implements Transporter {
   public Car() {}
   public void drive() {
      //code elided
   }
   public void move() {
       this.drive();
   }
}
```

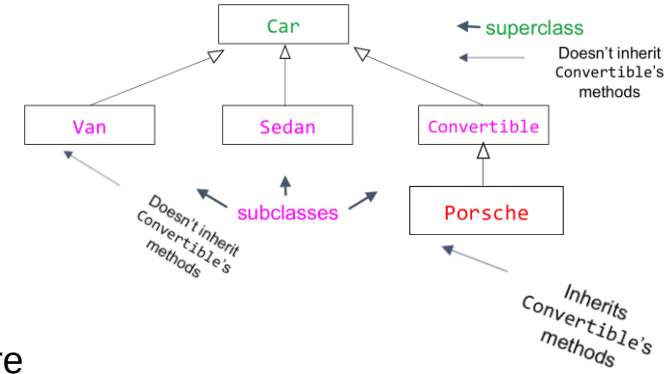
```
public class Bike implements Transporter {
   public Bike() {}
   public void pedal() {
       //code elided
   }
   public void move() {
       this.pedal();
   }
}
```

Inheritance and Polymorphism

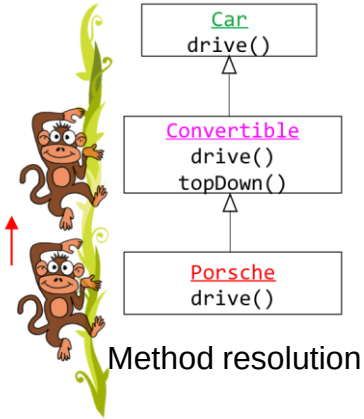
Inheritance

- 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();
    }
}
```

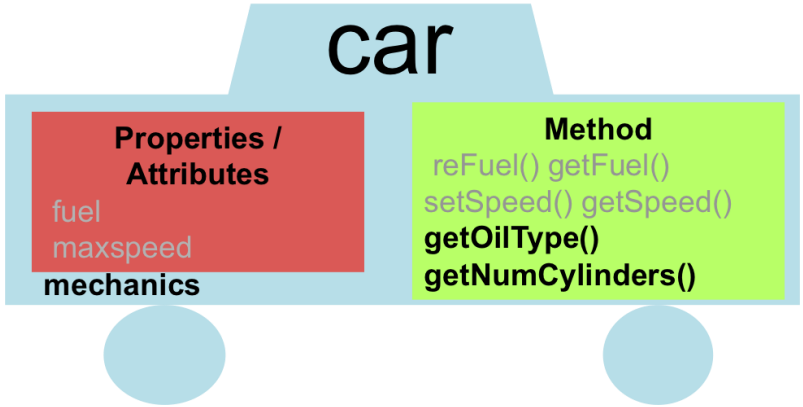
```
public class Van extends Car{
    @Override
    public void loadPassengers(){
        Passenger p1 = new Passenger();
        p1.sit();
        Passenger p2 = new Passenger();
        p2.sit();
        Passenger p3 = new Passenger();
        p3.sit();
    }
}
```

- 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

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*.

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();
}
```

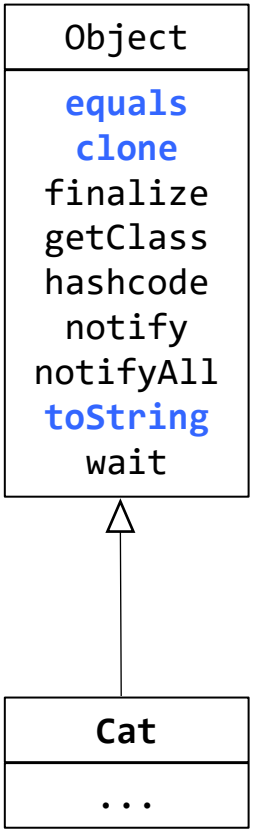
Mechanics cannot be extended as it is declared as final

```
public class ModifiedMechanics extends Mechanics {

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

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.



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`

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!

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
  - ……

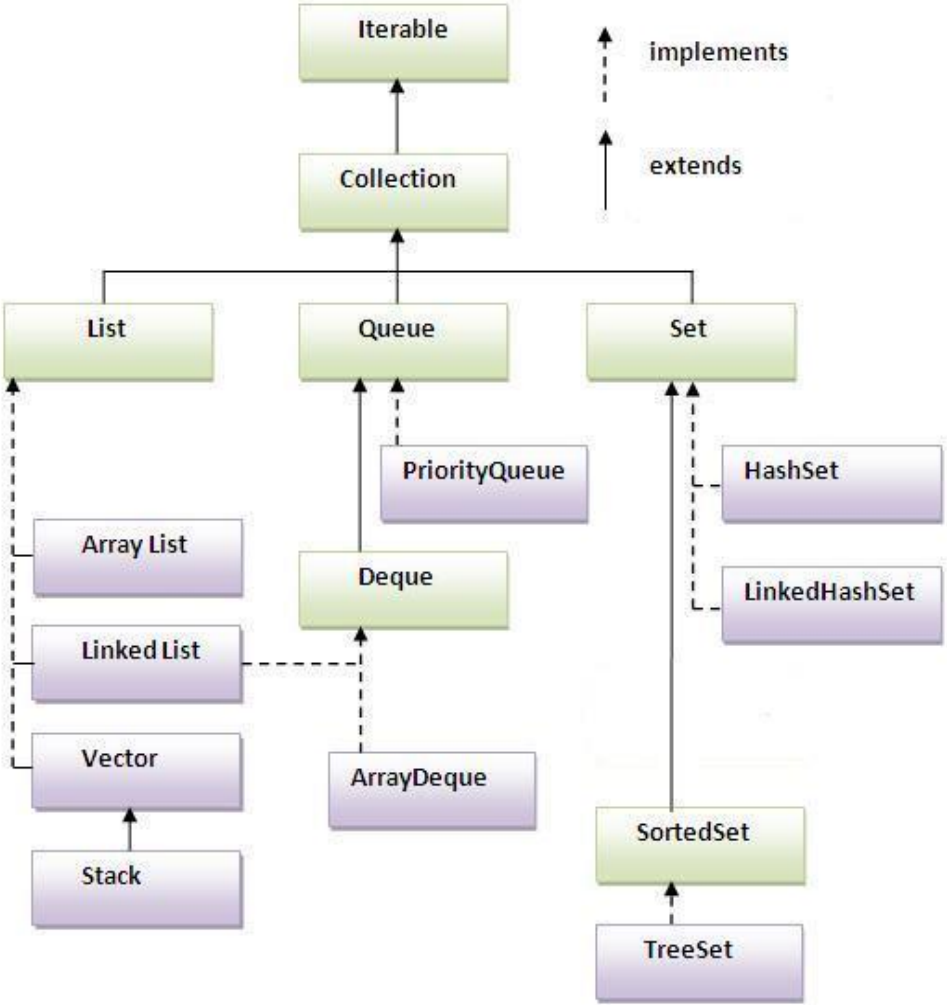
Cup == Generic Container

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 primate 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



Java Collection Framework

- A collection (sometimes called a *container*) is simply an object that groups multiple elements into a single unit
  - Iterator interface provides access to the content of a collection
  - Collection interface defines fundamental methods that are enough to define the basic behavior of a collection
  - Lists are like resizable arrays
    - ArrayList and LinkedList
  - Set interface methods are same as Collection interface but it does not allow duplicates
    - HashSet and TreeSet
  - Maps keep unique <key, value> pairs
    - HashMap and TreeMap

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);
            }
        }
    }
}
```

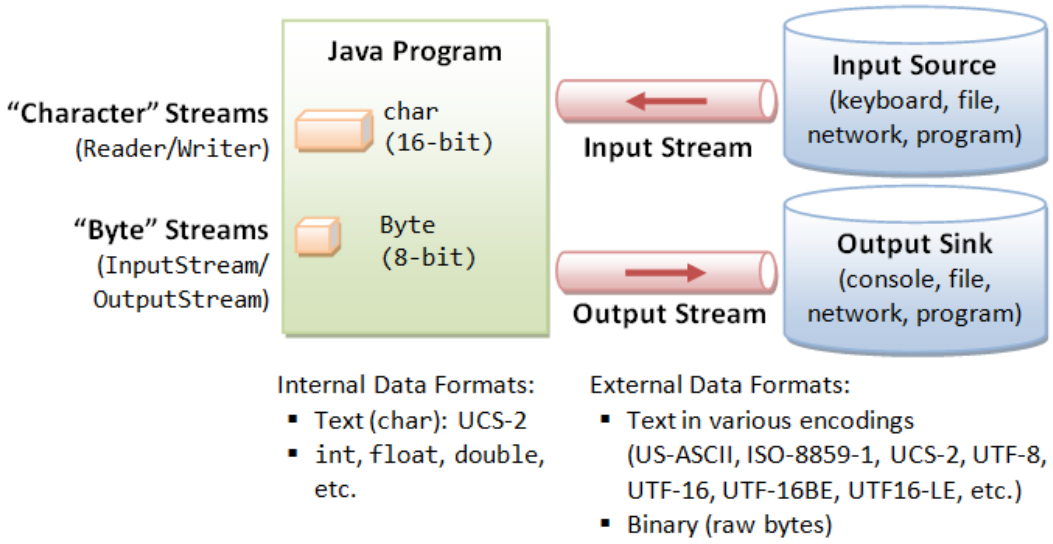
```
public class Andy {
    .....
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NullPointerException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() {
        try {
            _water = _wendy.getADrink();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

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`

I/O Streams

- Stream is a sequence of data
- Flows in/out the program to/from an external source such as file, network, console, etc.
- Types
  - Byte stream
    - Low level I/O (binary files)
  - Character stream
    - Processing text files



- Reading

```
open a stream
while more information
    read information
close the stream
```

- Writing

```
open a stream
while more information
    write information
close the stream
```

Combining Streams into Chains

```
public static void main(String args[])
                        throws IOException
{
    Scanner in = null;
    PrintWriter out = null;
    try {
        in = new Scanner( new BufferedReader( new
                            FileReader("input.txt")));
        out = new PrintWriter( new
                            FileWriter("output.txt"));
        while (in.hasNext()) {
            out.println(in.next());
        }
    } finally {
        if (in != null)
            in.close();
        if (out != null)
            out.close();
    }
}
```

- Here we are combining three classes for breaking input into tokens:
  - Scanner
  - BufferedReader
  - FileReader
- BufferedReader will read one line at a time and Scanner will be able to parse this line by white space separated tokens

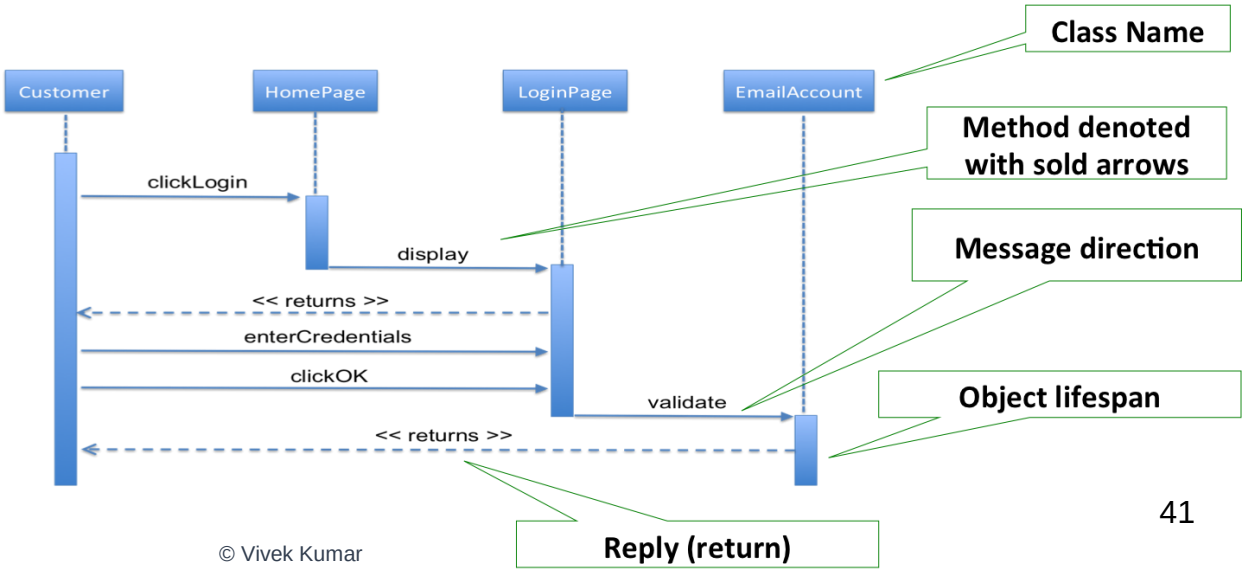
What is UML?

- UML stands for Unified Modeling Language
- **It's used to analyze, design, and implement software-based systems**
- We need a modeling language to:
  - help develop efficient, effective and correct designs, particularly Object Oriented designs
  - communicate clearly with project stakeholders (concerned parties: developers, customer, etc)
  - give us the "big picture" view of the project

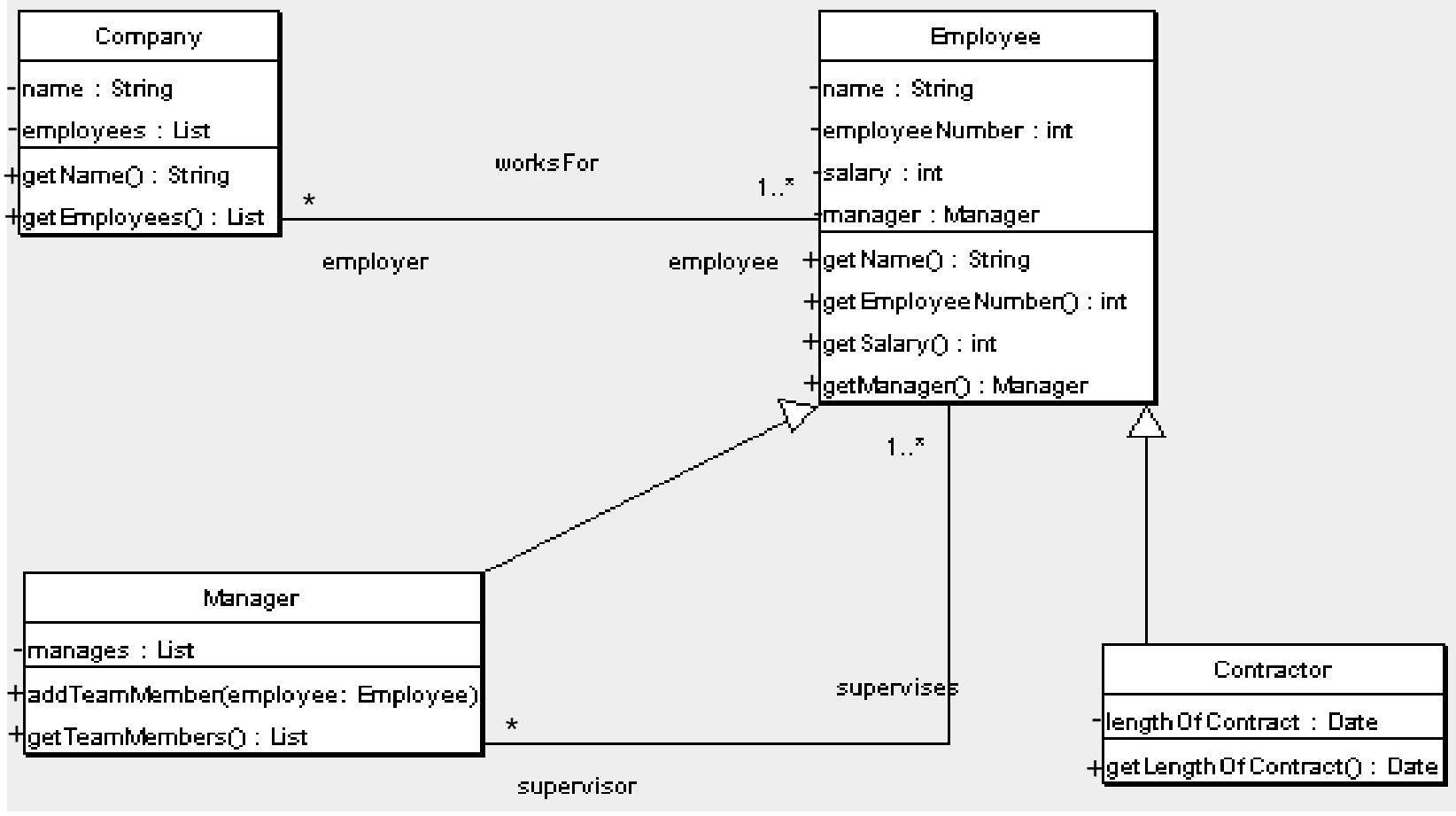
UML Diagrams

Three types of UML diagrams that we will cover:

1. **Class diagrams:** Represents static structure
2. **Use case diagrams:** Sequence of actions a system performs to yield an observable result to an actor
3. **Sequence diagrams:** Shows how groups of objects interact in some behavior



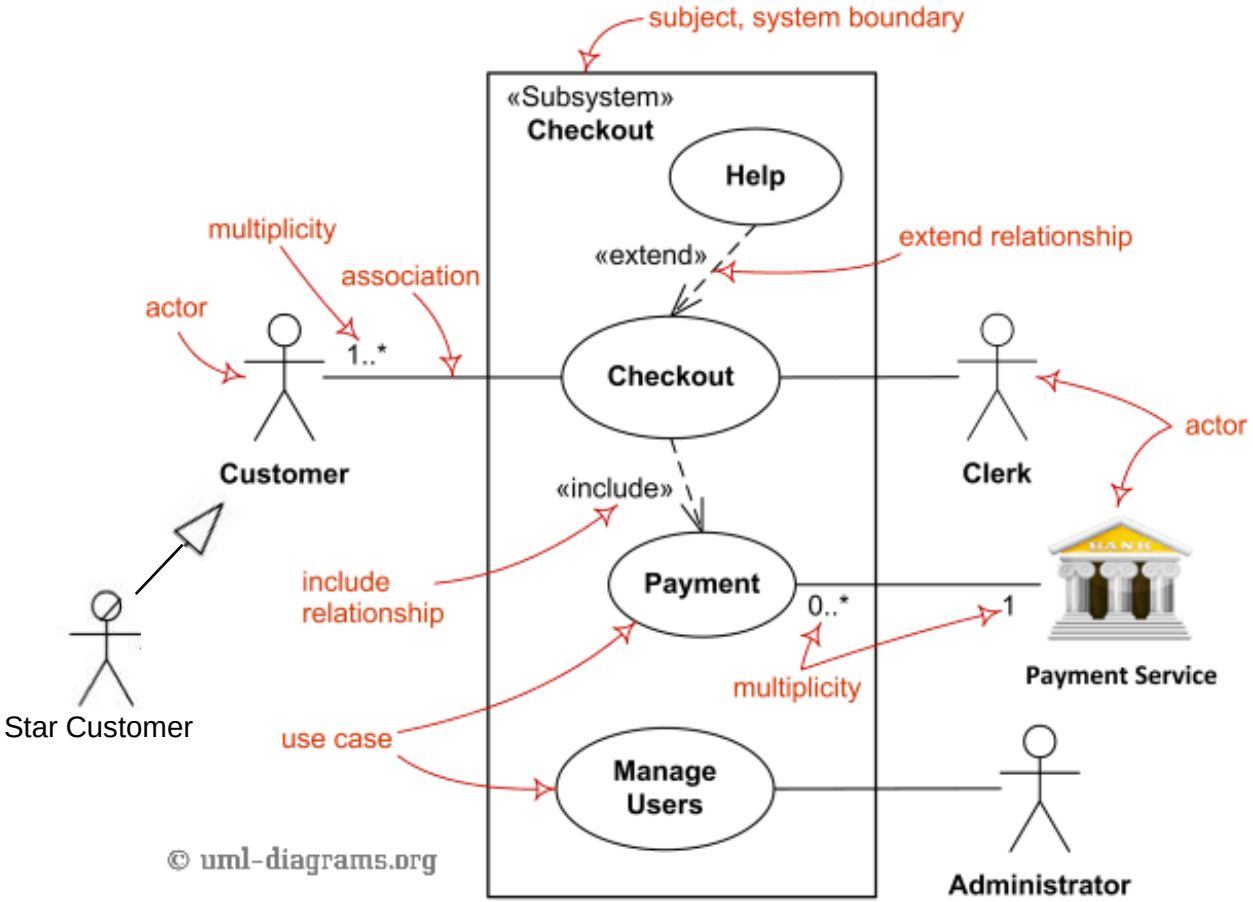
UML Class Diagram: Static Structure Diagram



UML Use Cases

- **Use case diagrams** describe what a system does from the standpoint of an external observer. **The emphasis is on *what* a system does rather than *how***
- Document interactions between user(s) and the system
  - User (actor) is not part of the system itself
  - But an actor can be *another* system

Sample Use Case Diagram



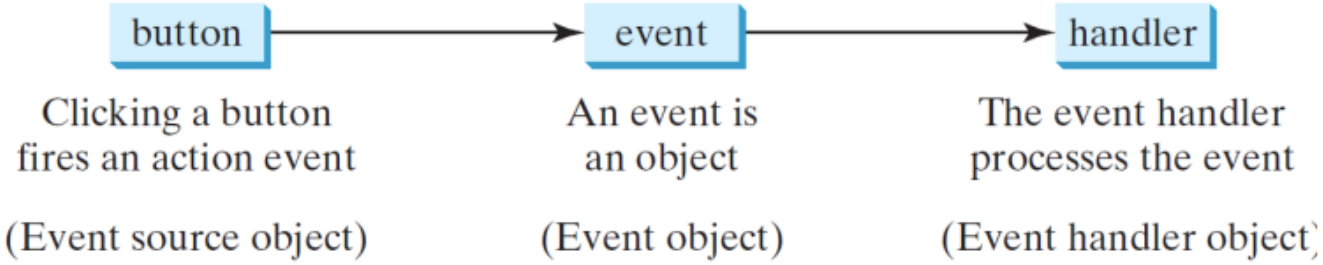
JavaFX Application Life Cycle

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    //Override the start method in the Application class
    @Override
    public void start(Stage primaryStage) {
        // Set the stage title
        primaryStage.setTitle("MyJavaFX");
        // Create a button and place it in the scene
        Button btn = new Button(”Hello World");
        Scene scene = new Scene(btn, 200, 250);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }

}
```

1. Constructs an instance of the specified Application class
2. Calls the concrete method init()
3. Calls start(javafx.stage.Stage) method (must be Overridden)
4. Waits for the application to finish
5. Calls the concrete method stop()

How to Handle GUI Events

- Source object: button
  - An event is generated by external user actions such as mouse movements, mouse clicks, or keystrokes
- An event can be defined as a type of signal to the program that something has happened
- Listener object contains a method for processing the event.

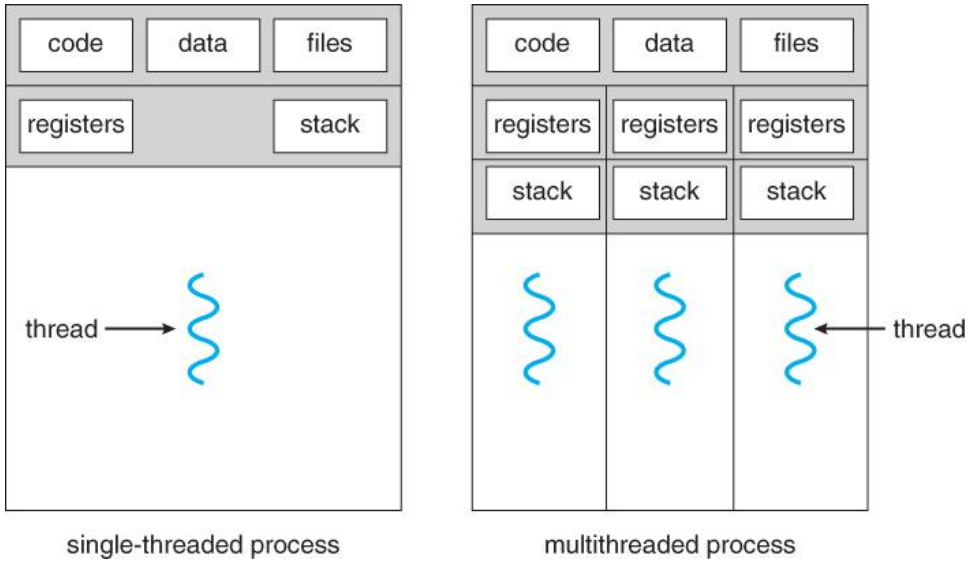


Example: Event Programming

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button(“Say Hello World”);

        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}
```

- Using **anonymous** inner classes for creating listener objects
  - It combines declaring an inner class and creating an instance of the class in one step
  - An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit extends or implements clause
    - An anonymous inner class must implement all the abstract methods in the superclass or in the interface
  - An anonymous inner class always uses the no-arg constructor from its superclass to create an instance

Processes and Threads



single-threaded process

multithreaded process

- Processes are heavyweight
  - Personal address space (allocated memory)
  - Communication across process always requires help from Operating System
- Threads are lightweight
  - Share resources inside the parent process (code, data and files)
    - Easy to communicate across sibling threads!
  - They have their own personal stack (local variables, caller-callee relationship between function)
    - Each thread is assigned a different job in the program
- A process can have one or more threads

Creating Threads in Java

- There are two ways to create your own **Thread** object
  - Implementing the **Runnable** interface
  - Subclassing the **Thread** class and instantiating a new object of that class

- In both cases the **run()** method should be implemented

Parallel Array Sum By Implementing Runnable Interface

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = left.getResult() + right.getResult();
    }
}
```

- Implement `java.lang.Runnable` interface
- Implement the method “public void run()”
- Create two threads (t1 & t2)
  - t1 will calculate the sum of left half of the array and t2 will calculate the sum of right half of array
    - Before creating t1 and t2 we must create objects of Runnable type that should be passed to the Thread constructor
- Start both the threads by calling the start() method in Thread class
- Wait for both the threads to complete their execution by calling join() method
- Sum the partial results from each threads to get the final results

Parallel Array Sum By Subclassing Thread

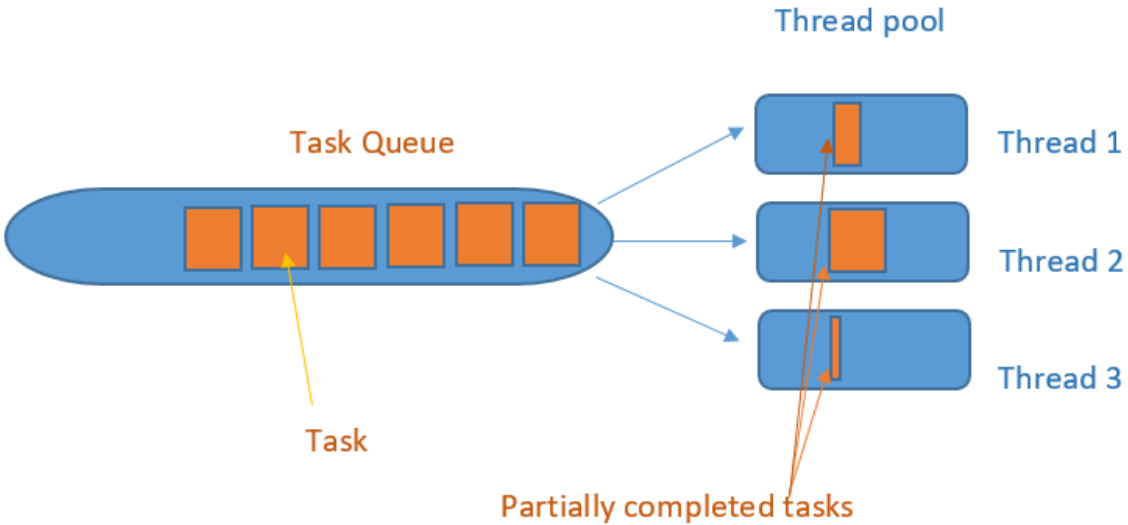
```
public class ArraySum extends Thread {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    @Override
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                              throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum t1 = new ArraySum(array, 0, size/2);
      ArraySum t2 = new ArraySum(array, size/2, size);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = t1.getResult() + t2.getResult();
    }
}
```

- Only three changes are required
  1. Instead of implementing Runnable, now the ArraySum class will extend Thread class
  2. Override the run() method as Thread class also has empty-body implementation of run()
  3. ArraySum objects are themselves Thread objects and hence now no need to explicitly call constructor of Thread class

Runnable v/s Subclassing Thread

- **Multiple inheritance is not allowed in Java** hence if our ArraySum class extends Thread then it cannot extend any other class. By implementing Runnable our ArraySum can easily extend any other class
- **Subclassing is used in OOP to add additional feature**, modifying or improving behavior. If no modifications are being made to Thread class then use Runnable interface
- **Thread can only be started once**. Runnable is better as same object could be passed to different threads
- If just run() method has to be provided then **extending Thread class is an overhead for JVM**

Introduction to Thread-Pool



- Thread-pool consists of a fixed number of threads
  - Provided by the Java runtime
- User application creates “task” rather than threads
- These tasks are added to a task-pool
- Free threads from thread-pool takes out a task from task-pool and execute it

Fig. Source: http://www.geek-programmer.com/what-are-thread-pools-in-java/

Parallel Array Sum Using Java ExecutorServices

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ExecutorService exec = Executors.newFixedThreadPool(2);
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      exec.execute(left); exec.execute(right);
      if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L, TimeUnit.SECONDS);
      }
      int result = left.getResult() + right.getResult();
    }
}
```

- An `ExecutorService` is a group of thread objects (thread pool), each running some variant of the following
  - `while (....) { get work and run it; }`
- `ExecutorService` methods:
  - `isTerminated`
    - Returns true if all tasks are terminated following the shutdown
  - `awaitTermination`
    - Blocks until all tasks have completed execution after a shutdown request
- Important that you wait for all tasks to terminate after a shutdown request

Topic-6: Mutual Exclusion

Mutual Exclusion

- ***Critical section:*** a block of code that access shared modifiable data or resource that should be operated on by only one thread at a time
- ***Mutual exclusion:*** a property that ensures that a critical section is only executed by a thread at a time.
  - *Otherwise it results in a race condition!*

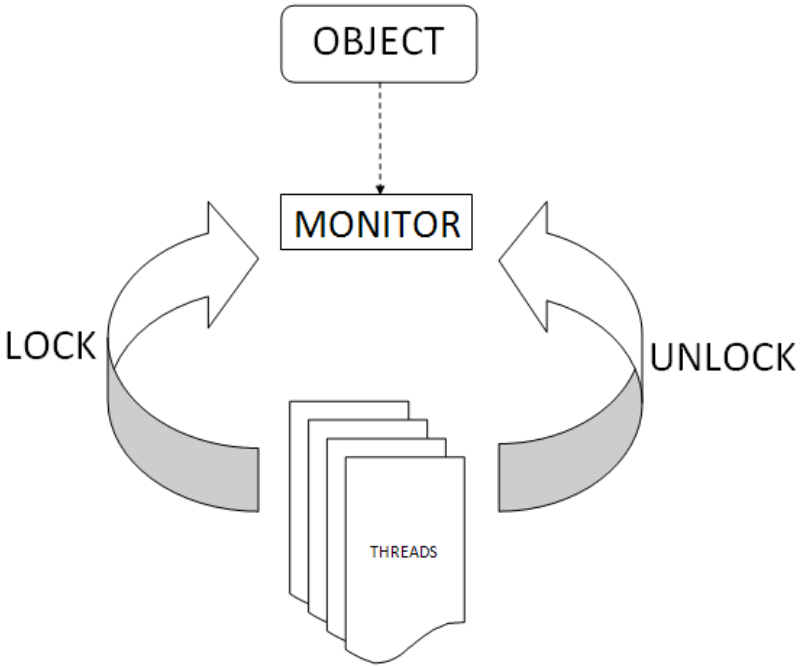
Acknowledgement: Slides adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit

Implementing Mutual Exclusion

```
class Counter implements Runnable {
    volatile int counter = 0;
    // Both the versions of run method below is correct
    public synchronized void run() { counter++; }
 /* public void run() { synchronized(this) {counter++;} } */
    public static void main(String[] args)
                            throws InterruptedException {
        ExecutorService exec =
                    Executors.newFixedThreadPool(2);

        Counter task = new Counter();
        for(int i=0; i<1000; i++) {
            exec.execute(task);
        }

        if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L,TimeUnit.SECONDS);
        }

        System.out.println(task.counter);
    }
}
```

- **Critical section**
  - The **synchronized** methods (or block) define the critical sections
  - By using **synchronized** keyword we achieved mutual exclusion
- **volatile** keyword for avoiding memory consistency issues
  - For faster data access, memory referenced by a CPU is first copied from main memory (RAM) onto its local cache
  - The updated memory content on cache is not immediately written back to RAM

Monitors



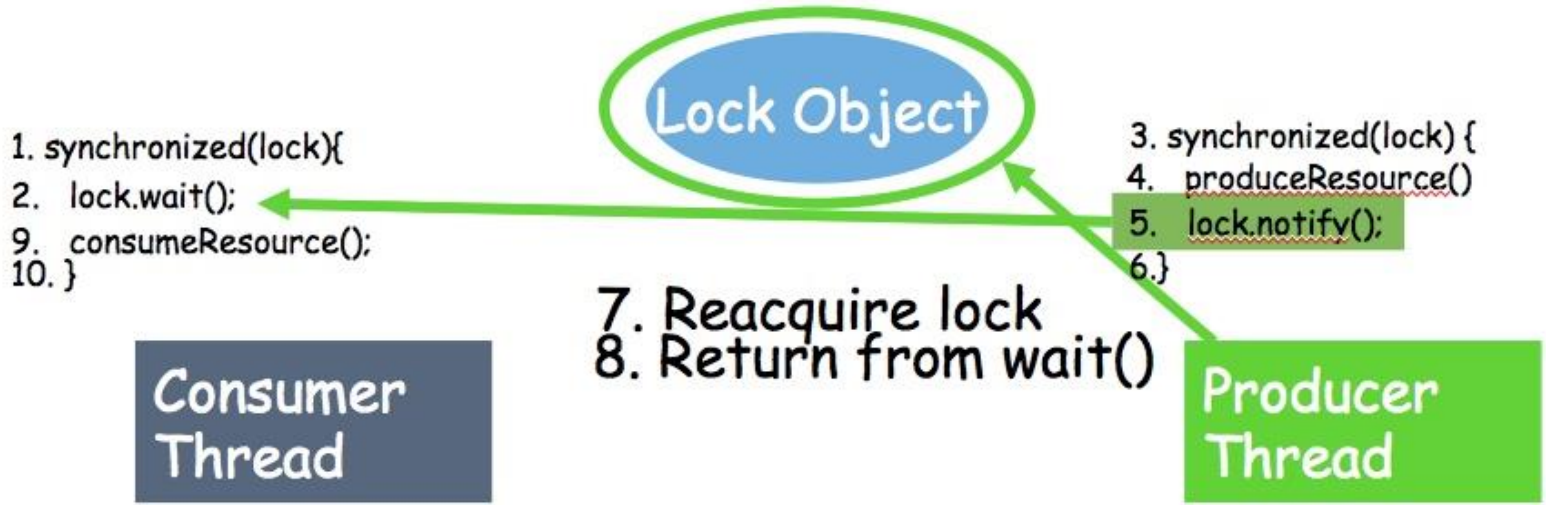
- Each object has a “**monitor**” that is a token used to determine which application **thread** has control of a particular **object** instance
- In execution of a synchronized method (or block), access to the object monitor (lock) must be gained before the execution
- Access to the object monitor is queued
- Demerits
  - Does not guarantee fairness
    - Lock might not be given to the longest waiting thread
  - Might lead to starvation
    - A thread can indefinitely hold the monitor lock for doing some big computation while other threads keep waiting to get this monitor lock
    - Not possible to interrupt the waiting thread
    - Not possible for a thread to decline waiting for the lock if its unavailable

Demerits of Monitor Lock

- Does not guarantee fairness
  - Lock might not be given to the longest waiting thread
- Might lead to starvation
  - A thread can indefinitely hold the monitor lock for doing some big computation while other threads keep waiting to get this monitor lock
  - Not possible to interrupt the thread who owns the lock
  - Not possible for a thread to decline waiting for the lock if its unavailable

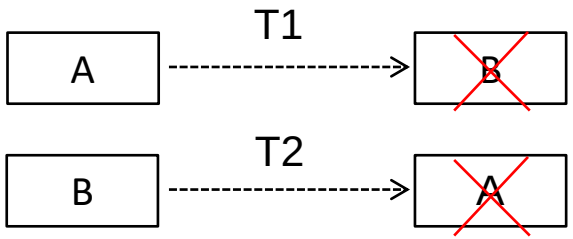
Producer Consumer Application Using Wait/Notify

- The `wait()` method is part of the class **java.lang.Object**
- It requires a lock on the object's monitor to execute
- It must be called from a synchronized method, or from a synchronized segment of code
- `wait()` causes the current thread to relinquish the CPU and wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object
- Upon call for wait(), the thread releases ownership of this monitor and waits until another thread notifies the waiting threads of the object



Deadlock Avoidance

```
class NEFTtransfer {
    Account A, B;
    int amount;
    // prone to deadlock
    void run() {
      synchronized(A) { // A locked
         synchronized(B) { // B locked
            A.debit(amount);
            B.credit(amount);
         } // B unlocked
      } // A unlocked
    }
}
```



- Deadlock occurs when multiple threads need the same set of locks but obtain them in different order
- Deadlock avoidance
  - Lock ordering
    - Ensure that all locks are taken in same order by any thread
      - E.g., in the code on left, first sort both the lock objects (e.g. based on account id of “A” and “B” accounts) and then always lock in a particular order followed by unlock in reverse order

I hope you enjoyed the course..

All the best for your end semester exam and final project deadline!