CSE201: Advanced Programming

Lecture 25: Endterm Review-1

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

Today's (Review) Lecture

1. Generic programming
2. IO Streams
3. UML
4. Event driven programming

Topic-1: Generic Programming

Generics

- Enables types (classes and interfaces) to be parameters when defining classes, interfaces, and methods


```
public interface Pair <K, V> {
    public K getKey();
    public V getValue();
}

public class MyPair <K, V> implements Pair<K, V> {
    private K key;
    private V value;
    public MyPair(K k, V v) { key=k; value=v; }
    public K getKey() { return key; }
    public V getValue() { return value; }

    public static void main(String[] args) {
        MyPair<Integer, String> p1 = new MyPair<Integer, String>(201, “Advanced
Programming”);
        MyPair<Integer, String> p2 = new MyPair<Integer, Integer>(0, 1);
        MyPair<Integer, String> p3 = new MyPair<Integer, Double>(0, 0.11);
    }
}
```

Motivation (1/3): Elimination of Casts

```
.......

public int countCoins(int value, List coins) {

    int total=0;
    Iterator it = coins.iterator();
    while(it.hasNext()) {
        Coin coin = (Coin) it.next();
        if(coin.denomination() == value) total++;
    }
    return total;
}

public static void main(String[] args) {

    List coins = new LinkedList();
    coins.add(new Coin(1));
    coins.add(new Coin(2));
    coins.add(new Coin(10));

    Coin lowest = (Coin) coins.get(0);
}
```

- If we follow this technique to code then we can't avoid the two typecasting shown here
  - Although its annoying but no other work around!

Motivation (2/3): Cleaner Code

```
.......
public int countCoins(int value, List<Coin> coins) {

    int total=0;
    for(Coin coin : coins) {
        if(coin.denomination() == value) total++;
    }
    return total;
}

public static void main(String[] args) {

    List<Coin> coins = new LinkedList<Coin>();
    coins.add(new Coin(1));
    coins.add(new Coin(2));
    coins.add(new Coin(10));

    Coin lowest = coins.get(0);
}
```

- However, using generic programming, we can avoid typecasting at all places!
- Also, we can iterate over the elements in a much cleaner way
- Generic programming is also called as “Parametric Polymorphism”

Motivation (3/3): Stronger Type Checks at Compilation

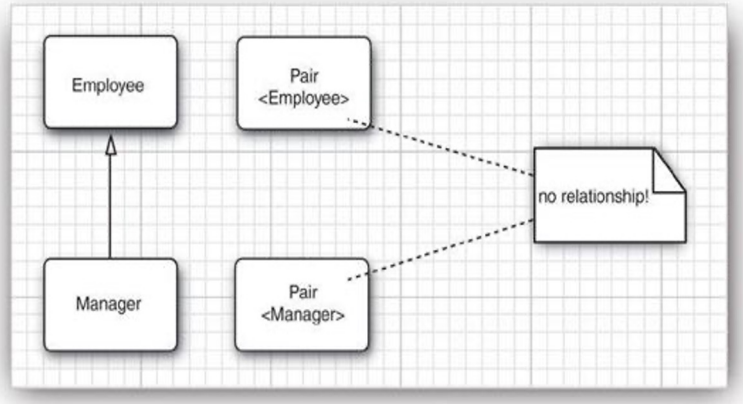
```
.....
public static void main(String[] args) {
    List coins = new LinkedList();
    // NO CHECK, UNSAFE!
    coins.add("ABCDEF");

    // RUNTIME ERROR!
    Coin lowest = (Coin) coins.get(0);
}
```

```
.....
public static void main(String[] args) {
    List<Coin> coins = new LinkedList<Coin>();
    // COMPILE TIME ERROR!
    coins.add("ABCDEF");

    // RUNTIME IS SAFE!
    Coin lowest = coins.get(0);
}
```

- Generic programming can help us locate errors at compile time rather than at runtime

Restrictions (Compile time checks)

1. Type parameters cannot be instantiated with primitive types
   - `MyGenericList <int> var = new MyGenericList<Integer>;`
2. Instantiating type variable is not allowed
   - `T my_var = new T();`
3. Cannot use instanceof with parameterized types
   - public <T> doSomething(List<T> list) {
     if(list instanceof ArrayList<Integer>) { …. }
     }
4. Type variables are not valid in static contexts of generic classes
   - `public static void doSomething(T a) { . `
5. Generic does not support sub-typing



Upper Bounded and Lower Bounded Wildcard

```
public class Main {
    .....
   static void print(ArrayList<? extends Car> list){
        .......
   }
   public static void main(String[] arg){
        .......
   }
}
```

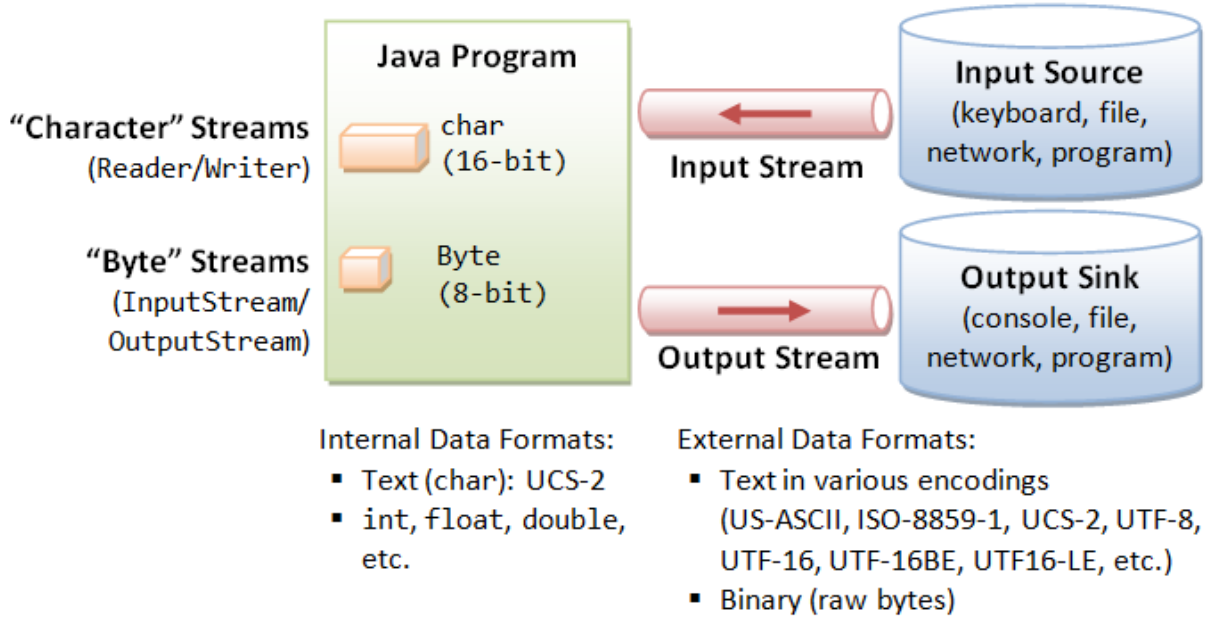
```
public class Main {
    .....
  static void print(ArrayList<? super Integer> list){
        .......
  }
  public static void main(String[] arg){
        .......
  }
}
```

- Upper bounded wildcard
  - Here the print method will only accept ArrayList of Car type or its subclass type
- Lower bounded wildcard
  - Here the print method will only accept ArrayList of Integer or any Type that is supertype of Integer
    - Integer
    - Number
    - Object

Topic-2: I/O Streams

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

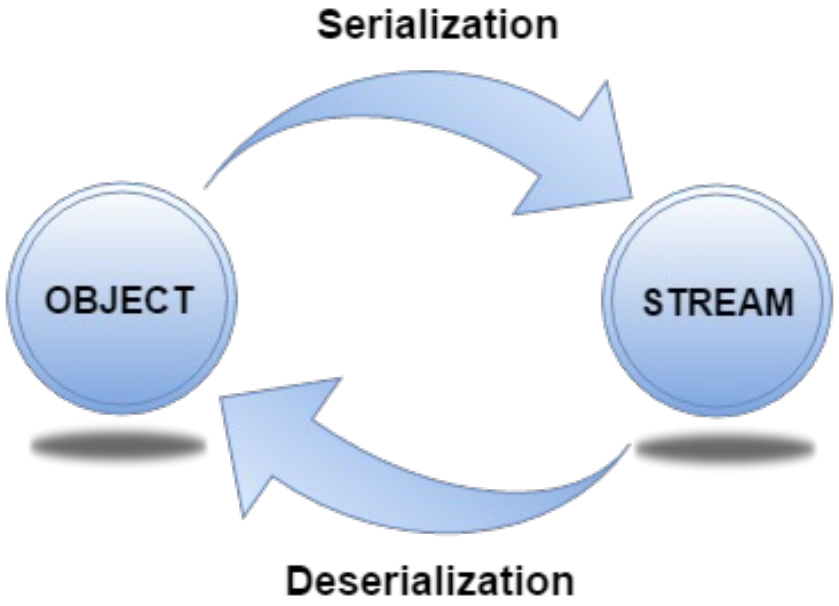
```
open a stream
while more information
    write information
close the stream
```

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

Serialization and Deserialization



- Serialization in Java is a mechanism of writing the state of an **object** into a **byte stream**
  - **Note:** it's the object state that is recorded but not the actual class definition ("class file")
- The reverse operation is called deserialization
- Some usage
  - Storing live objects in a file
  - Hibernating applications
  - Moving object state over the network (marshaling)

Example: Serializing and Deserializing

```
1. import java.io.*;
2.
3. class Manager implements Serializable {
4.     private String name;
5.     public Manager(String n) { ..... }
6. }

7. public class Main {
8.    public static void serialize()
9.                          throws IOException {
10.        Manager s1 = new Manager("Amy");
11.        ObjectOutputStream out = null;
12.        try {
13.           out = new ObjectOutputStream (
14.                 new FileOutputStream("out.txt"));
15.           out.writeObject(s1);
16.        } finally {
17.           out.close();
18.        }
19.   }
20.
/* Continued on RHS window */
```

```
   /* Continued from LHS window */
21.    public static void deserialize()
22.       throws IOException, ClassNotFoundException {
23.        ObjectInputStream in = null;
24.        try {
25.            in =  new ObjectInputStream (
26.                   new FileInputStream("out.txt"));
27.            Manager s1 = (Manager) in.readObject();
29.        } finally {
30.            in.close();
31.        }
32.    }
33.
34.    public static void main(String[] args)
35.        throws IOException,ClassNotFoundException {
36.
37.       serialize();
39.       deserialize();
40.    }
41. } /* End of Main class */
```

Suppose you have a Client.java that only has the above deserializes() method. Compilation of Client.java will generate two class files Client.class and Manager.class. If you try running "java Client" without Manager.class in its classpath then ClassNotFoundException will be thrown at Line 27 above.

Rules for Serialization

- Classes must implement Serializable interface
- All instance type fields in the class should be Serializable
  - Otherwise, `NotSerializableException` will be thrown at runtime
- If parent class implements Serializable interface, subclass need not do
  - Otherwise, parent class must have a default constructor. Not doing so will generate `InvalidClassException`
- Static fields do not represents object state but they represent class state, hence no point in serializing them
- If you don't want any instance type field to be serialized, then mark that as "`transient`"
- It is always advisable to declare "`serialVersionUID` " in each serializable class with your own number of choice
  - The class declaration might have got updated (e.g. added new fields) after serialization and now deserializing the object will generate `InvalidClassException`

Topic-3: UML

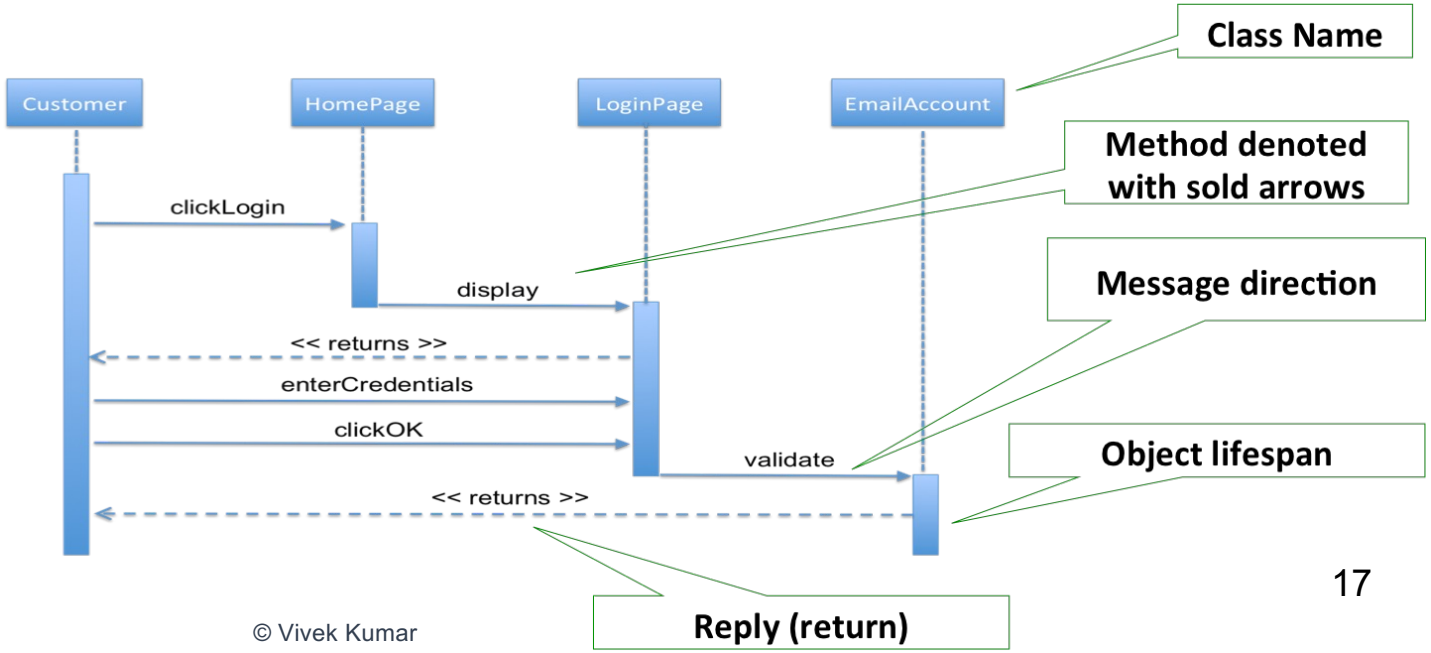
What is UML?

- 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

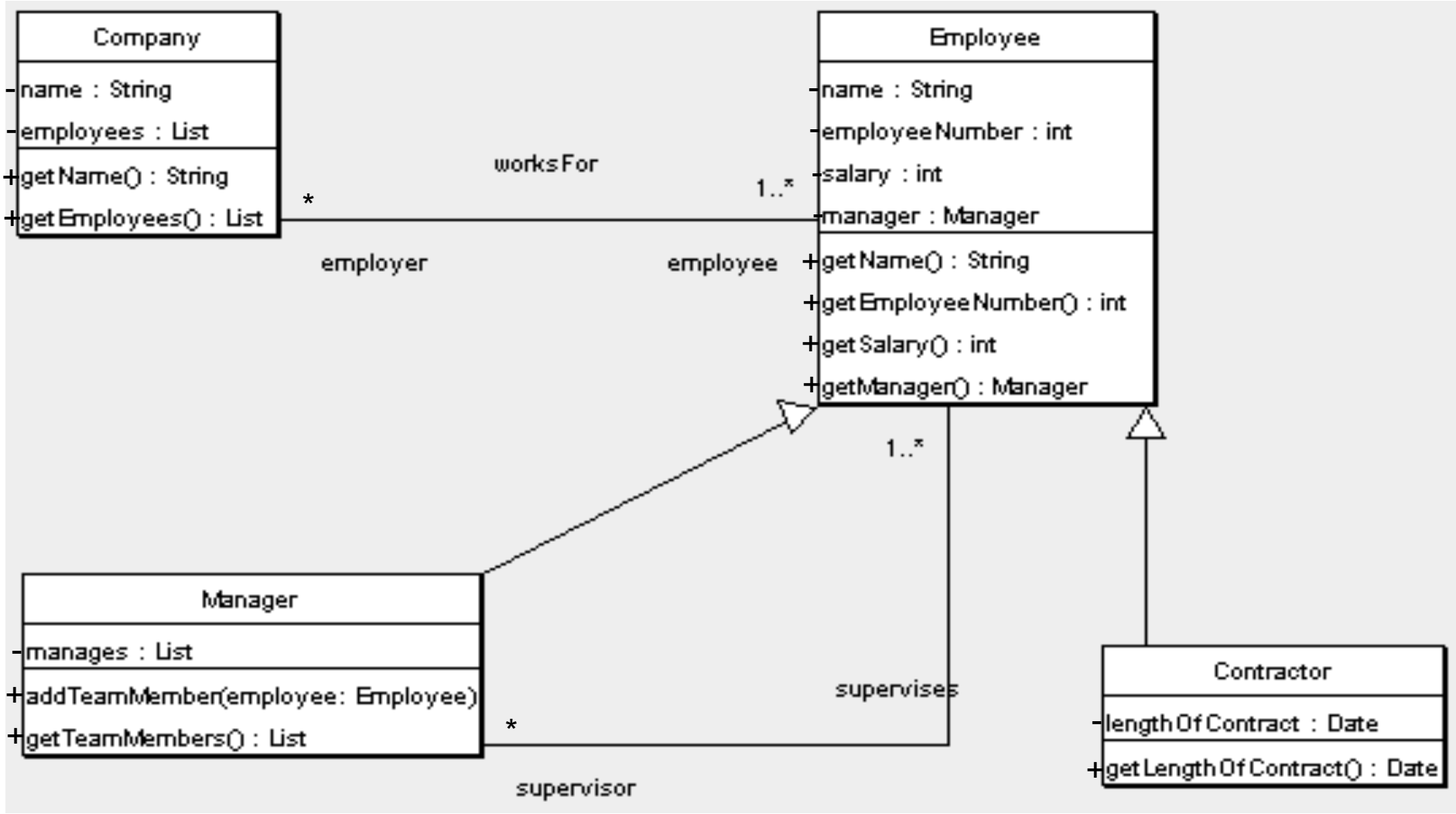
UML Diagrams

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



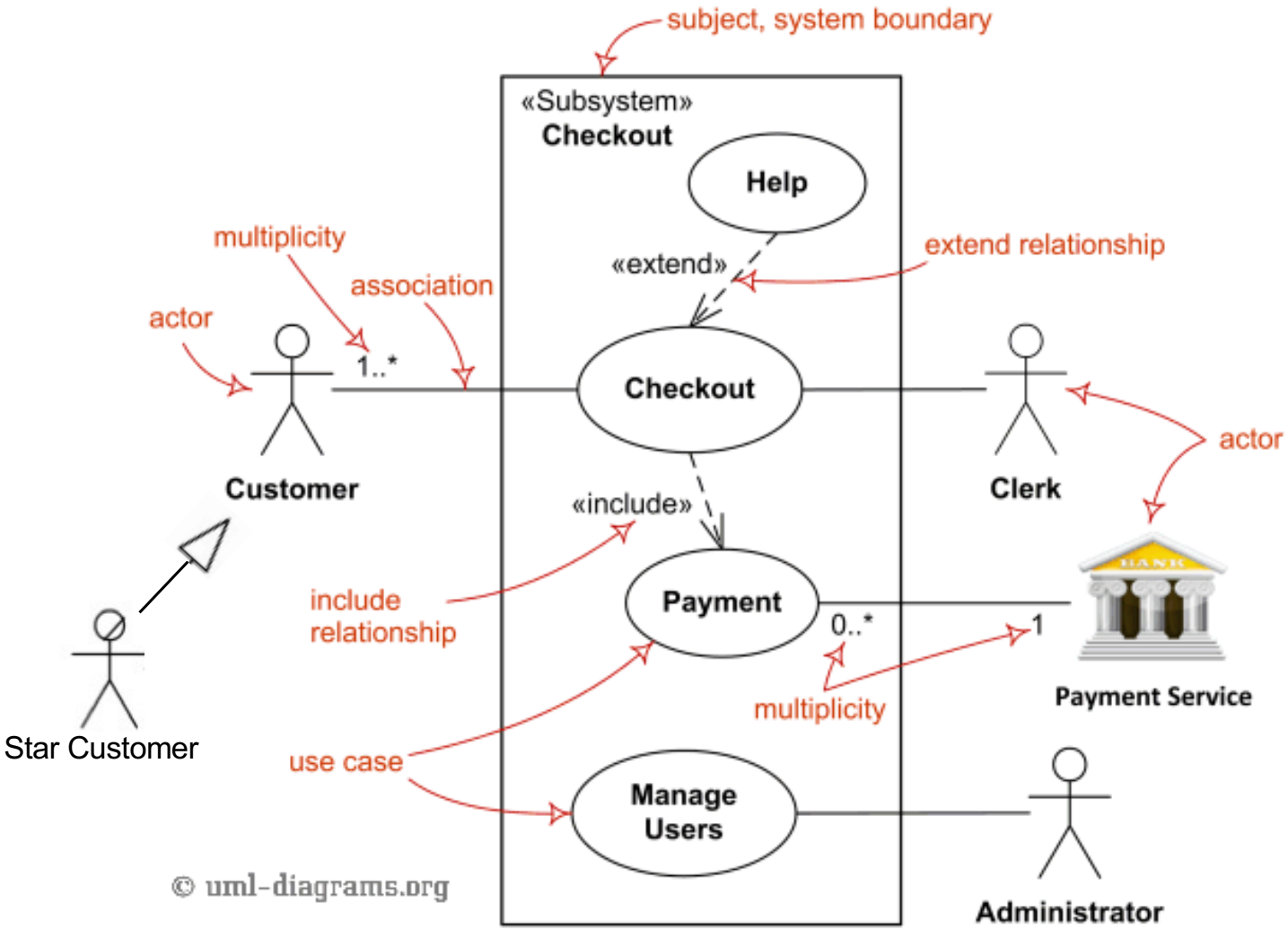
UML Class Diagram: Static Structure Diagram



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



Topic-4: Event Driven Programming using JavaFX

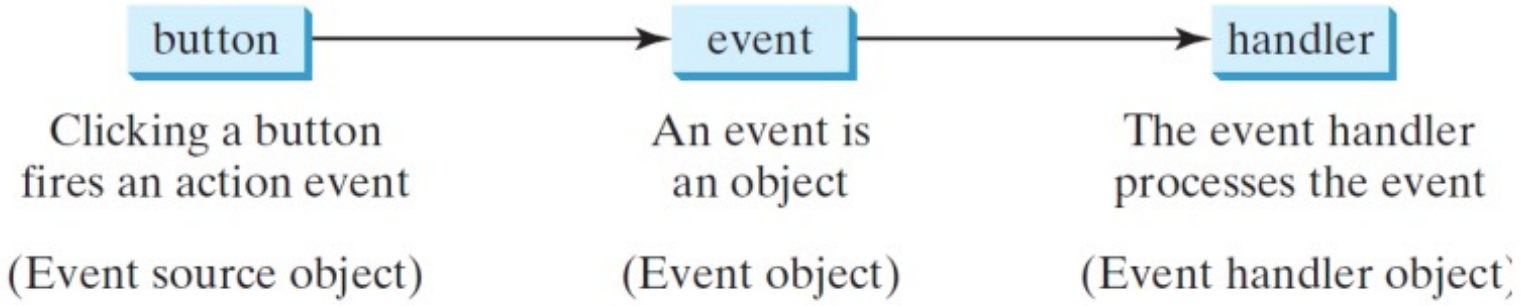
JavaFX Application Life Cycle

```
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

Next Lecture

- Endterm review lecture-2 (Last remaining lecture)
  - Multithreading
  - Mutual exclusion
  - No review on design pattern as we recently completed it, and also no review on inheritance, interfaces, and polymorphism as we covered these topics extensively while discussing design patterns
- **Bonus quiz**
  - **Entire syllabus**