Objectives

- In this session, you will learn to:
  - Compile and run Java program
  - Explore Java class loader
  - Explore the concept of garbage collection
  - Use encapsulation in Java class design
  - Model business problems using Java classes
  - Make classes immutable
  - Create and use Java subclasses

How to Compile and Run

- `javac` compiler:
  - Used to compile a source file
  - Syntax:
    ```
    javac -cp <path to other classes> -d
    <complier output path> <path to
    source>.java
    ```
- Java interpreter:
  - Called `java`
  - Used to execute java program
  - Syntax:
    ```
    java -cp <path to other classes> <package
    name>.<classname>
    ```

How to Compile and Run (Contd.)

- Java compiler and interpreter use the `CLASSPATH` environment variable to locate `.class` file.
- Setting `CLASSPATH`:
  - Use the `CLASSPATH` environment variable

    or
  - Use `-cp` command line switch

Java Class Loader

- Java Virtual Machine (JVM):
  - Loads the compiled Java class files.
  - Uses the `java.lang.ClassLoader` class.
- The `-verbose` flag is used during execution to see the working of the class loader.
- The following code snippet explains the concept of Java class loader:

```
public class Test {
public void someOperation(){
      Employee e = new Employee();
        //...
}}

Test.class.getClassLoader().loadClass("Employee"
);
```

The class loader is called to load this class into memory

Garbage Collection

- Garbage Collection (GC):
  - Is also known as automatic memory management.
  - Is the automatic recycling of dynamically allocated memory.
  - Is performed by the garbage collector that resides in the JVM.
- In Java, the `new` keyword is used to dynamically allocate memory to the object.

Encapsulation

- Encapsulation:
  - Means to enclose in a capsule.
  - Wraps the internal workings of a Java object.
  - Hides the data variables from the user of the object.
  - Enables methods to provide an explicit service to the user of the object but hides the implementation.

Encapsulation: Example

- What data and operations would you encapsulate in an object that represents an employee?

Employee ID
Name
Social Security Number
Salary

Set Name
Raise Salary

Encapsulation: Private Data, Public Methods

- To hide implementation details:
  - Declare all fields of class as `private`.
  - Declare all the methods as `public`.
- The following embedded Word document shows how to implement encapsulation in a class.

Public and Private Access Modifiers

- The `public` keyword:
  - Applied to fields and methods
  - Allows any class in any package to access the fields or methods
- The `private` keyword:
  - Applied to fields and methods
  - Allows access only to other methods within the class itself

Revisiting Employee

- To implement encapsulation in the `Employee` class, make fields `private`, as shown in the following code snippet:

```
package come.example.model;
public class Employee
{
    private int empId;
    private String name;
    private String ssn;
    private double salary;
   //... constructor and methods
}
```

Encapsulation step 1:
Hide the data (fields).

Method Naming: Best Practices

- To implement encapsulation in methods:
  - Hide as many of the implementation details as possible.
  - Name the method such that it clearly identifies its use.

Model Business Problems Using Java Classes

- Model business problems:
  - The `Employee` class uses setters for `ID`, `salary`, and `SSN` fields.
  - Its fields should not change.
  - To refine the `Employee` class:
    - Remove the all the setter methods and define only the `setName()` and `raiseSalary()` methods.
- The following embedded Word document shows the refined version of the `Employee` class.

Make Classes as Immutable as Possible

- The `Employee` class does not have the setter methods, to initialize the `ID`, `salary`, and `SSN` fields.
- Replace the `no-arg` constructor with parameterized constructor to ensure that the `Employee` class is immutable.
- The following code snippet can be used to replace the `no-arg` constructor of the `Employee` class:

```
public Employee (int empId, String name,
                 String ssn, double
salary)
{
        this.empId = empId;
        this.name = name;
        this.ssn = ssn;
        this.salary = salary;
   }
```

Creating Subclasses

- Subclasses:
  - Define a new class in terms of an existing class.
  - Are created from an existing class, with added features, to implement specialization.
  - Consider the existing class as the superclass.
  - Inherit the non-private members of the superclass.
  - Can access all the non-private methods of the superclass.
  - Can define new methods to add more functionality.
  - Do not allow the new methods to be accessible to the superclass objects.
  - Use the `extends` keyword to inherit a superclass.
  - Help in reducing redundant code from the application.

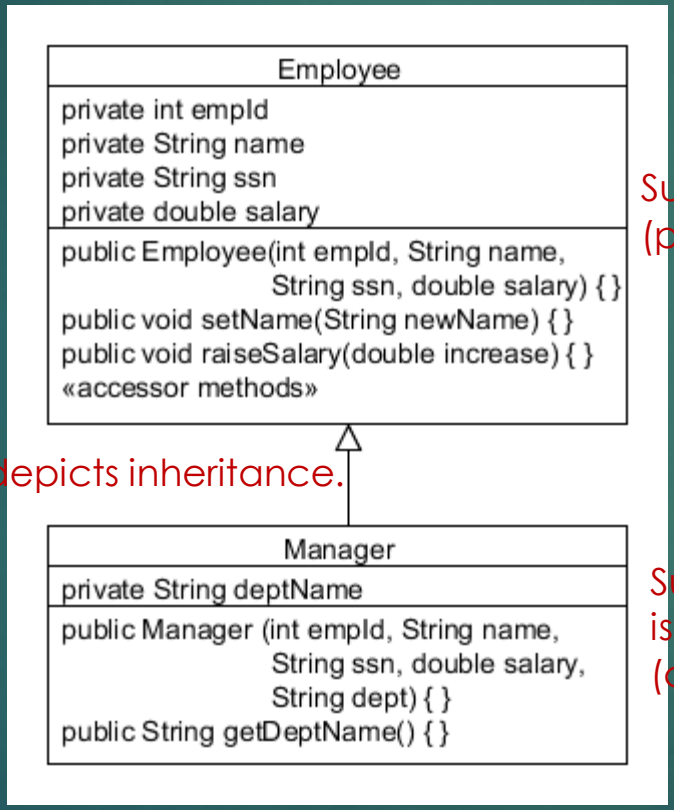
Creating Subclasses (Contd.)

- The following code snippet shows the `Manager` class that is also an `Employee`:

```
package com.example.domain;
    public class Manager {
    private int empId;
    private String name;
    private String ssn;
    private double salary;
    private String deptName;
// access and mutator methods...
}
```

Creating Subclasses (Contd.)

- The following diagram shows the concept of subclasses using the `Employee` and the `Manager` classes.



Constructors Are Not Inherited

- Constructors:
  - Are not inherited.
  - Ways to get them:
    - Write your own constructor.
    - Use the default constructor.

Using super in Constructors

- The `super` keyword:
  - Is used to call the constructor of the parent class.
  - Must be the first statement of the constructor.
  - If not provided, a default call to `super()` is inserted by the compiler.
  - May also be used to invoke a method or access the (non-private) field of the parent class.

Constructing a Manager Object

- The following code snippet shows how to create an object of the `Manager` class:

```
Manager mgr = new Manager
(102, "Barbara Jones",
"107-99-9078",
109345.67,Marketing");
```