3

Encapsulation and Subclassing

Objectives

After completing this lesson, you should be able to do the following:

- Use encapsulation in Java class design
- Model business problems by using Java classes
- Make classes immutable
- Create and use Java subclasses
- Overload methods

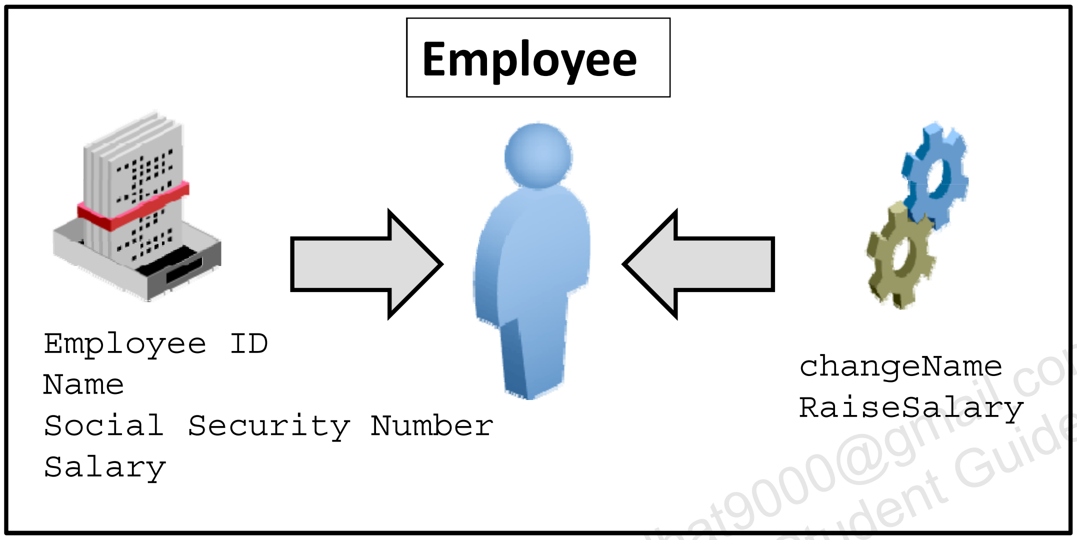
Encapsulation

- Encapsulation is one of the four fundamental object-oriented programming concepts. The other three are inheritance, polymorphism, and abstraction.
- The term *encapsulation* means to enclose in a capsule, or to wrap something around an object to cover it.
- Encapsulation covers, or wraps, the internal workings of a Java object.
  - Data variables, or fields, are hidden from the user of the object.
  - Methods, the functions in Java, provide an explicit service to the user of the object but hide the implementation.
  - As long as the services do not change, the implementation can be modified without impacting the user.

An analogy for encapsulation is the steering wheel of a car. When you drive a car, whether it is your car, a friend's car, or a rental car, you probably never worry about how the steering wheel implements a right-turn or left-turn function. The steering wheel could be connected to the front wheels in a number of ways: ball and socket, rack and pinion, or some exotic set of servo mechanisms. As long as the car steers properly when you turn the wheel, the steering wheel encapsulates the functions you need—you do not have to think about the implementation.

Encapsulation: Example

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



A Simple Model

Suppose that you are asked to create a model of a typical employee. What data might you want to represent in an object that describes an employee?

- **Employee ID:** You can use this as a unique identifier for the employee.
- **Name:** Humanizing an employee is always a good idea.
- **Social Security Number:** For United States employees only. You may want some other identification for non-U.S. employees.
- **Salary:** How much the employee makes is always good to record.

What operations might you allow on the employee object?

- **Change Name:** If the employee gets married or divorced, there could be a name change.
- **Raise Salary:** It increases based on merit.

After an employee object is created, you probably do not want to allow changes to the Employee ID or Social Security fields. Therefore, you need a way to create an employee without alterations except through the allowed methods.

Encapsulation: Public and Private Access Modifiers

- The `public` keyword, applied to fields and methods, allows any class in any package to access the field or method.
- The `private` keyword, applied to fields and methods, allows access only to other methods within the class itself.

```
Employee emp=new Employee();
emp.salary=2000; // Compiler error- salary is a private field
emp.raiseSalary(2000); //ok
```

- The `private` keyword can also be applied to a method to hide an implementation detail.

Java has three visibility modifiers: `public`, `private`, and `protected`.

Encapsulation: Private Data, Public Methods

One way to hide implementation details is to declare all of the fields `private`.

- The `Employee` class currently uses `public` access for all of its fields.
- To encapsulate the data, make the fields `private`.

```
public class Employee {

    private int empId;
    private String name;
    private String ssn;
    private double salary;

 //... constructor and methods
}
```

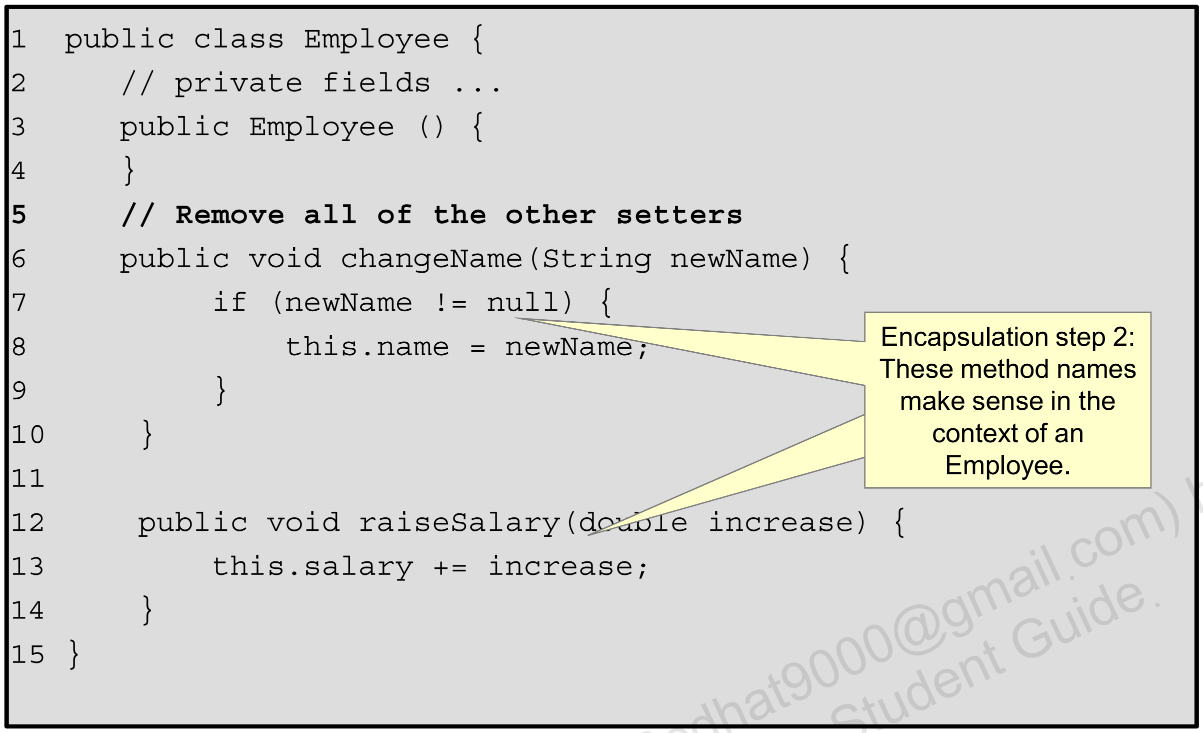
Declaring fields `private` prevents direct access to this data from a class instance.

```
// illegal!
emp.salary =
1_000_000_000.00;
```

In Java, we accomplish encapsulation through the use of visibility modifiers. Declaring Java fields private makes it invisible outside of the methods in the class itself.

In this example, the fields `custID`, `name`, and `amount` are now marked private, making them invisible outside of the methods in the class itself.

`Employee` Class Refined



```
public class Employee {
    // private fields ...
    public Employee () {
    }
    // Remove all of the other setters
    public void changeName(String newName) {
        if (newName != null) {
            this.name = newName;
        }
     }

     public void raiseSalary(double increase) {
        this.salary += increase;
     }
}
```

Encapsulation step 2: These method names make sense in the context of an Employee.

The current setter methods in the class allow any class that uses an instance of `Employee` to alter the object's ID, salary, and SSN fields. From a business standpoint, these are not operations you would want on an employee. Once the employee is created, these fields should be immutable (no changes allowed).

The `Employee` model as defined in the slide titled "Encapsulation: Example" had only two operations: one for changing an employee name (as a result of a marriage or divorce) and one for increasing an employee's salary.

To refine the `Employee` class, the first step is to remove the setter methods and create methods that clearly identify their purpose. Here there are two methods, one to change an employee name (`setName`) and the other to increase an employee salary (`raiseSalary`).

Note that the implementation of the `setName` method tests the string parameter passed in to make sure that the string is not a null. The method can do further checking as necessary.

Make Classes as Immutable as Possible

```
public class Employee {
    // private fields ...
    // Create an employee object
    public Employee (int empId, Strin
                          String ssn, double salary) {
        this.empId = empId;
        this.name = name;
        this.ssn = ssn;
       this.salary = salary;
     }

     public void changeName(String newName) { ... }

     public void raiseSalary(double increase) { ... }
}
```

Encapsulation step 3: Remove the no-arg constructor; implement a constructor to set the value of all fields.

Good Practice: Immutability

Finally, because the class no longer has setter methods, you need a way to set the initial value of the fields. The answer is to pass each field value in the construction of the object. By creating a constructor that takes all of the fields as arguments, you can guarantee that an `Employee` instance is fully populated with data *before* it is a valid employee object. This constructor *replaces* the default constructor.

Granted, the user of your class could pass null values, and you need to determine if you want to check for those in your constructor. Strategies for handling those types of situations are discussed in later lessons.

Removing the setter methods and replacing the no-arg constructor also guarantees that an instance of `Employee` has immutable Employee ID and Social Security Number (SSN) fields.

Method Naming: Best Practices

Although the fields are now hidden by using `private` access, there are some issues with the current `Employee` class.

- The setter methods (currently `public` access) allow any other class to change the ID, SSN, and salary (up or down).
- The current class does not really represent the operations defined in the original `Employee` class design.
- Two best practices for methods:
  - Hide as many of the implementation details as possible.
  - Name the method in a way that clearly identifies its use or functionality.
- The original model for the `Employee` class had a Change Name and an Increase Salary operation.

Choosing Well-Intentioned Methods

Just as fields should clearly define the type of data that they store, methods should clearly identify the operations that they perform. One of the easiest ways to improve the readability of your code (Java code or any other) is to write method names that clearly identify what they do.

Encapsulation: Benefits

The benefits of using encapsulation are as follows:

- Protects an object from unwanted access by clients
- Prevents assigning undesired values for its variables by the clients, which can make the state of an object unstable
- Allows changing the class implementation without modifying the client interface

Creating Subclasses

You created a Java class to model the data and operations of an `Employee`. Now suppose you wanted to specialize the data and operations to describe a `Manager`.

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

wait a minute...
this code looks very familiar....

ORACLE

Specialization by Using Java Subclassing

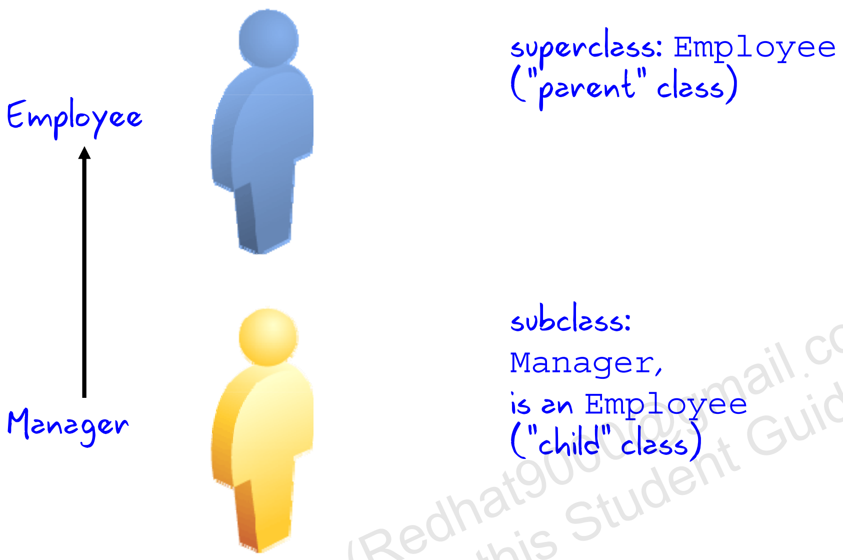
The `Manager` class shown here closely resembles the `Employee` class, but with some specialization. A `Manager` also has a department, with a department name. As a result, there are likely to be additional operations as well.

What this demonstrates is that a `Manager` is an `Employee`—but an `Employee` with additional features.

However, if we were to define Java classes this way, there would be a lot of redundant coding.

Subclassing

In an object-oriented language like Java, subclassing is used to define a new class in terms of an existing one.



A Simple Java Program

When an existing class is subclassed, the new class created is said to inherit the characteristics of the other class. This new class is called the *subclass* and is a specialization of the superclass. All the nonprivate fields and methods from the superclass are part of the subclass.

In this diagram, a `Manager` class gets all the nonprivate fields and all the public methods from `Employee`.

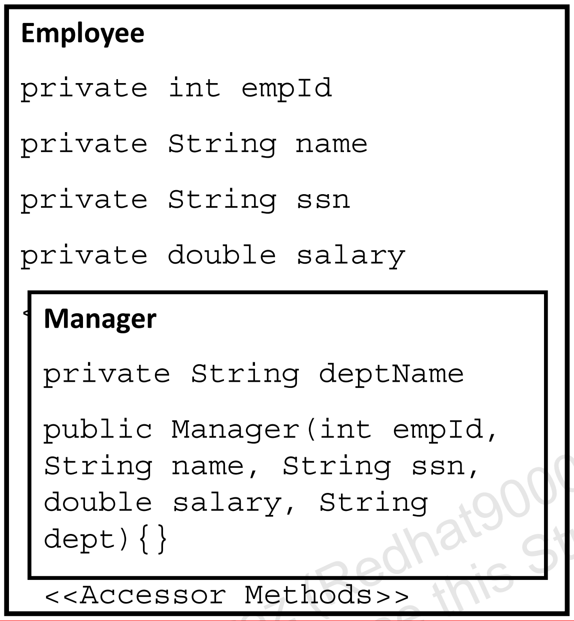
It is important to grasp that although `Manager` specializes `Employee`, a `Manager` is still an `Employee`.

Note: The term *subclass* is a bit of a misnomer. Most people think of the prefix “*sub*” as meaning “less.” However, a Java subclass is the sum of itself and its parent. When you create an instance of a subclass, the resulting in-memory structure contains all codes from the parent class, grandparent class, and so on all the way up the class hierarchy until you reach the class `Object`.

`Manager` Subclass

```
public class Manager extends Employee { }
```

The keyword **`extends`** creates the inheritance relationship:



The code snippet in the slide demonstrates the Java syntax for subclassing.

The diagram in the slide demonstrates an inheritance relationship between the `Manager` class and, its parent, the `Employee` class.

- The `Manager` class, by extending the `Employee` class, inherits all of the non-private data fields and methods from `Employee`.
- Since a manager is also an employee, then it follows that Manager has all of the same attributes and operations of Employee.

Note: The Manager class declares its own constructor. Constructors are *not* inherited from the parent class. There are additional details about this in the next slide.

Constructors in Subclasses

Although a subclass inherits all of the methods and fields from a parent class, it does not inherit constructors. There are two ways to gain a constructor:

- Write your own constructor.
- Use the default constructor.
  - If you do not declare a constructor, a default no-arg constructor is provided for you.
  - If you declare your own constructor, the default constructor is no longer provided.

Constructors in Subclasses

Every subclass inherits the nonprivate fields and methods from its parent (superclass). However, the subclass does not inherit the constructor from its parent. It must provide a constructor.

The *Java Language Specification* includes the following description:

"Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding."

Using `super`

To construct an instance of a subclass, it is often easiest to call the constructor of the parent class.

- In its constructor, `Manager` calls the constructor of `Employee`.
- The `super` keyword is used to call a parent's constructor.
- It must be the first statement of the constructor.
- If it is not provided, a default call to `super()` is inserted for you.
- The `super` keyword may also be used to invoke a parent's method or to access a parent's (nonprivate) field.

```
super (empId, name, ssn, salary);
```

Constructing a `Manager` Object

Creating a `Manager` object is the same as creating an `Employee` object:

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

- All of the `Employee` methods are available to `Manager`:

```
mgr.raiseSalary (10000.00);
```

- The `Manager` class defines a new method to get the Department Name:

```
String dept = mgr.getDeptName();
```

Even though the `Manager.java` file does not contain all of the methods from the `Employee.java` class (explicitly), they are included in the definition of the object. Thus, after you create an instance of a `Manager` object, you can use the methods declared in `Employee`.

You can also call methods that are specific to the `Manager` class as well.

Overloading Methods

Your design may call for several methods in the same class with the same name but with different arguments.

```
public void print (int i)
public void print (float f)
public void print (String s)
```

- Java permits you to reuse a method name for more than one method.
- Two rules apply to overloaded methods:
  - Argument lists *must* differ.
  - Return types *can* be different.
- Therefore, the following is not legal:

```
public void print (int i)
public String print (int i)
```

You might want to design methods with the same intent (method name), like `print`, to print out several different types. You could design a method for each type:

```
printInt(int i)
printFloat(float f)
printString(String s)
```

But this would be tedious and not very object-oriented. Instead, you can create a reusable method name and just change the argument list. This process is called *overloading*.

With overloading methods, the argument lists must be different—in order, number, or type. And the return types can be different. However, two methods with the same argument list that differ only in return type are not allowed.

Overloaded Constructors

- In addition to overloading methods, you can overload constructors.
- The overloaded constructor is called based upon the parameters specified when the `new` is executed.

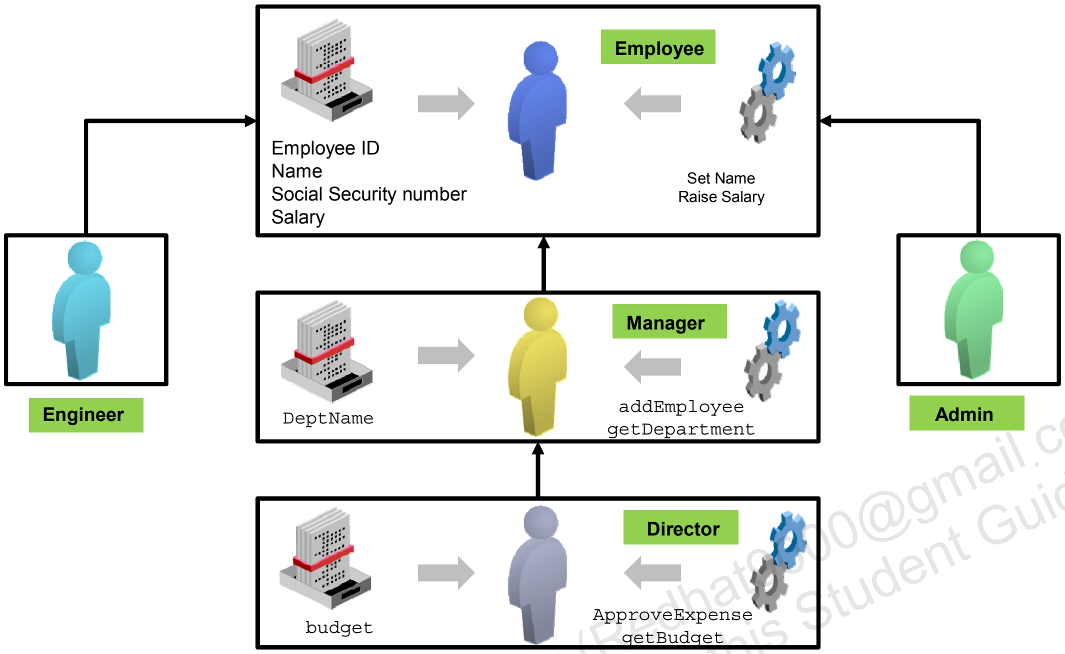
Overloaded Constructors: Example

```
public class Box {

    private double length, width, height;

    public Box() {
        this.length = 1;
        this.height = 1;
        this.width = 1;
    }

    public Box(double length) {
        this.width = this.length = this.height = length;
    }

    public Box(double length, double width, double height) {
        this.length = length;
        this.height = height;
        this.width = width;
        System.out.println("and the height of " + height + ".");
    }

    double volume() {
        return width * height * length;
    }
}
```

The example in the slide demonstrates overloaded constructors: there are three overloaded constructors, which vary based on the no of arguments.

Single Inheritance

The Java programming language permits a class to extend only one other class. This is called *single inheritance*.



Summary

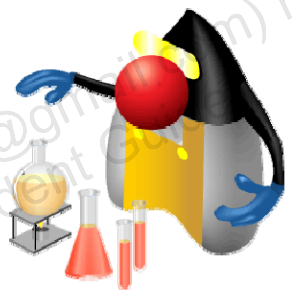
In this lesson, you should have learned how to:

- Use encapsulation in Java class design
- Model business problems by using Java classes
- Make classes immutable
- Create and use Java subclasses
- Overload methods

Practice 3-1 Overview: Creating Subclasses

This practice covers the following topics:

a. Applying encapsulation principles to the `Employee` class that you created in the previous practice
b. Creating subclasses of `Employee`, including `Manager`, `Engineer`, and Administrative assistant (`Admin`)
c. Creating a subclass of `Manager` called `Director`
d. Creating a test class with a `main` method to test your new classes



Quiz

Which of the following declarations demonstrates the application of good Java naming conventions?

a. `public class repeat { }`

b. `public void Screencoord (int x, int y){}`

c. `private int XCOORD;`

d. `public int calcOffset (int xCoord, int yCoord) { }`

Quiz

What changes would you perform to make this class immutable? (Choose all that apply.)

```
public class Stock {
    public String symbol;
    public double price;
    public int shares;
    public double getStockValue() { }
    public void setSymbol(String symbol) { }
    public void setPrice(double price) { }
    public void setShares(int number) { }
}
```

a. Make the fields `symbol`, `shares`, and `price` `private`.
b. Remove `setSymbol`, `setPrice`, and `setShares`.
c. Make the `getStockValue` method `private`.
d. Add a constructor that takes `symbol`, `shares`, and `price` as arguments.