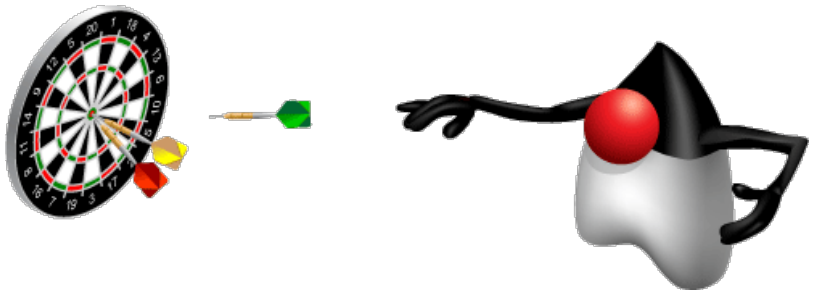
Encapsulation and Subclassing

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

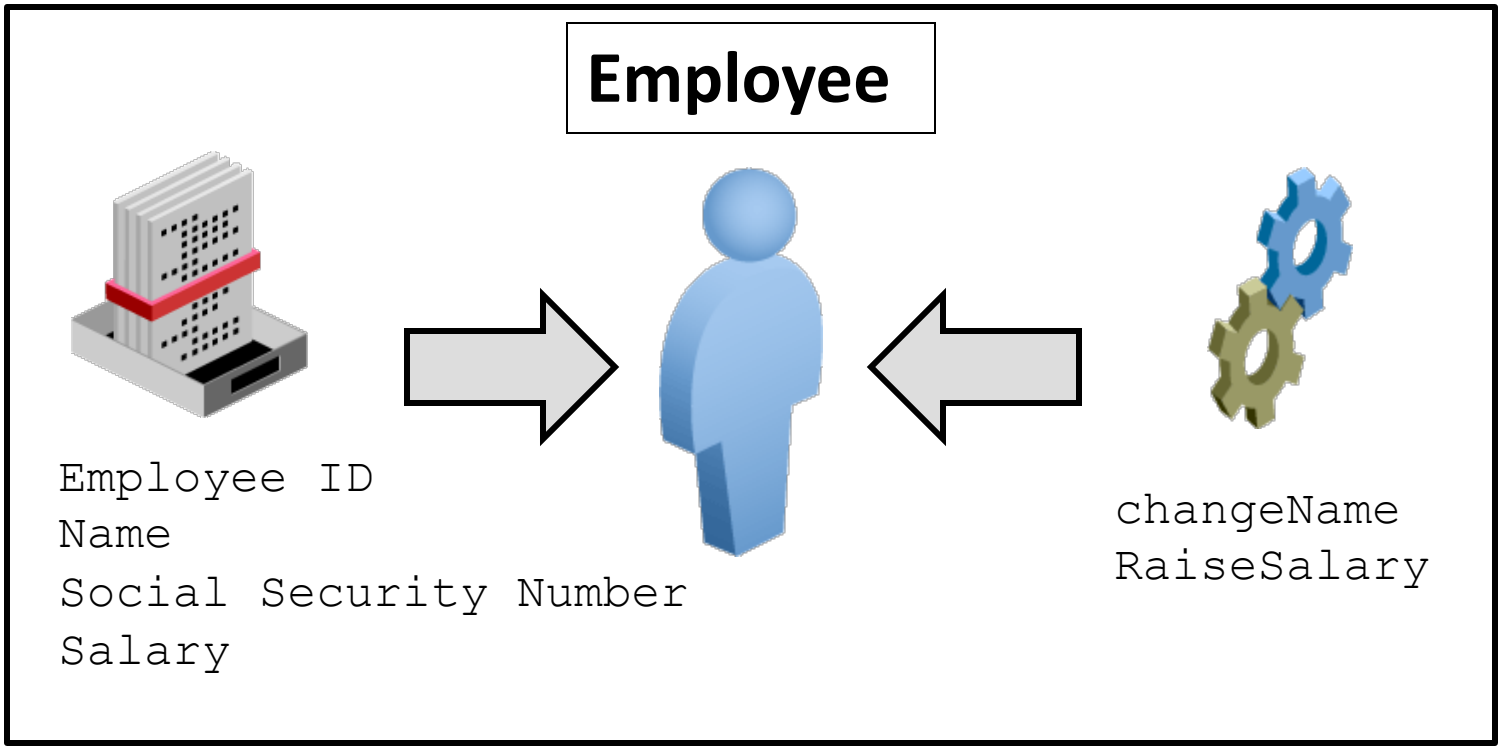


Encapsulation

- 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.

Encapsulation: Example

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



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.

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;
```

`Employee` Class Refined

```
1  public class Employee {
2      // private fields ...
3      public Employee () {
4      }
5      // Remove all of the other setters
6      public void changeName(String newName) {
7           if (newName != null) {
8               this.name = newName;
9           }
10      }
11
12      public void raiseSalary(double increase) {
13          this.salary += increase;
14      }
15 }
```

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

Make Classes as Immutable as Possible

```
1  public class Employee {
2     // private fields ...
3     // Create an employee object
4     public Employee (int empId, Strin
5                      String ssn, double salary) {
6         this.empId = empId;
7         this.name = name;
8         this.ssn = ssn;
9        this.salary = salary;
10     }
11
12     public void changeName(String newName) { ... }
13
14     public void raiseSalary(double increase) { ... }
15 }
```

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

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

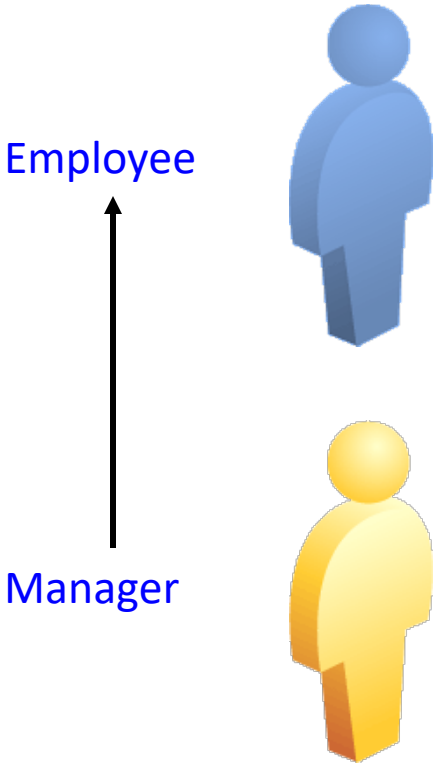
You created a Java class to model the data and operations of an `Employee`. Now suppose you wanted to specialize the data and operations to describe a `Manager`.

```
1  package com.example.domain;
2  public class Manager {
3      private int empId;
4      private String name;
5      private String ssn;
6      private double salary;
7      private String deptName;
8      public Manager () {  }
9      // access and mutator methods...
10 }
```

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

Subclassing

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



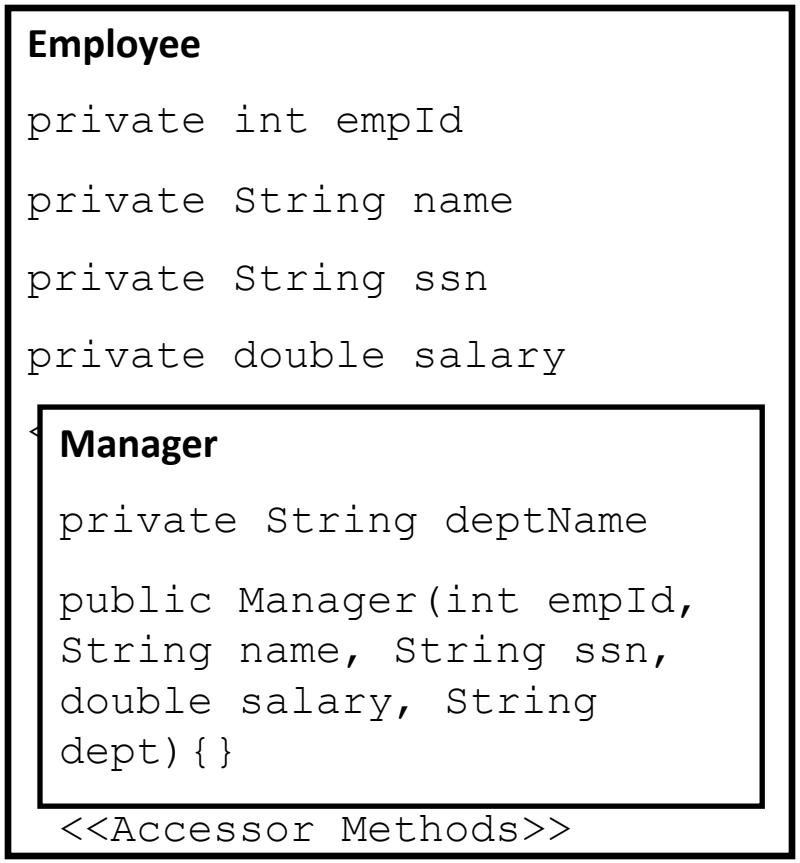
superclass: `Employee`
("parent" class)

subclass:
`Manager`,
is an `Employee`
("child" class)

`Manager` Subclass

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

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



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.

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();
```

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)
```

Overloaded Constructors

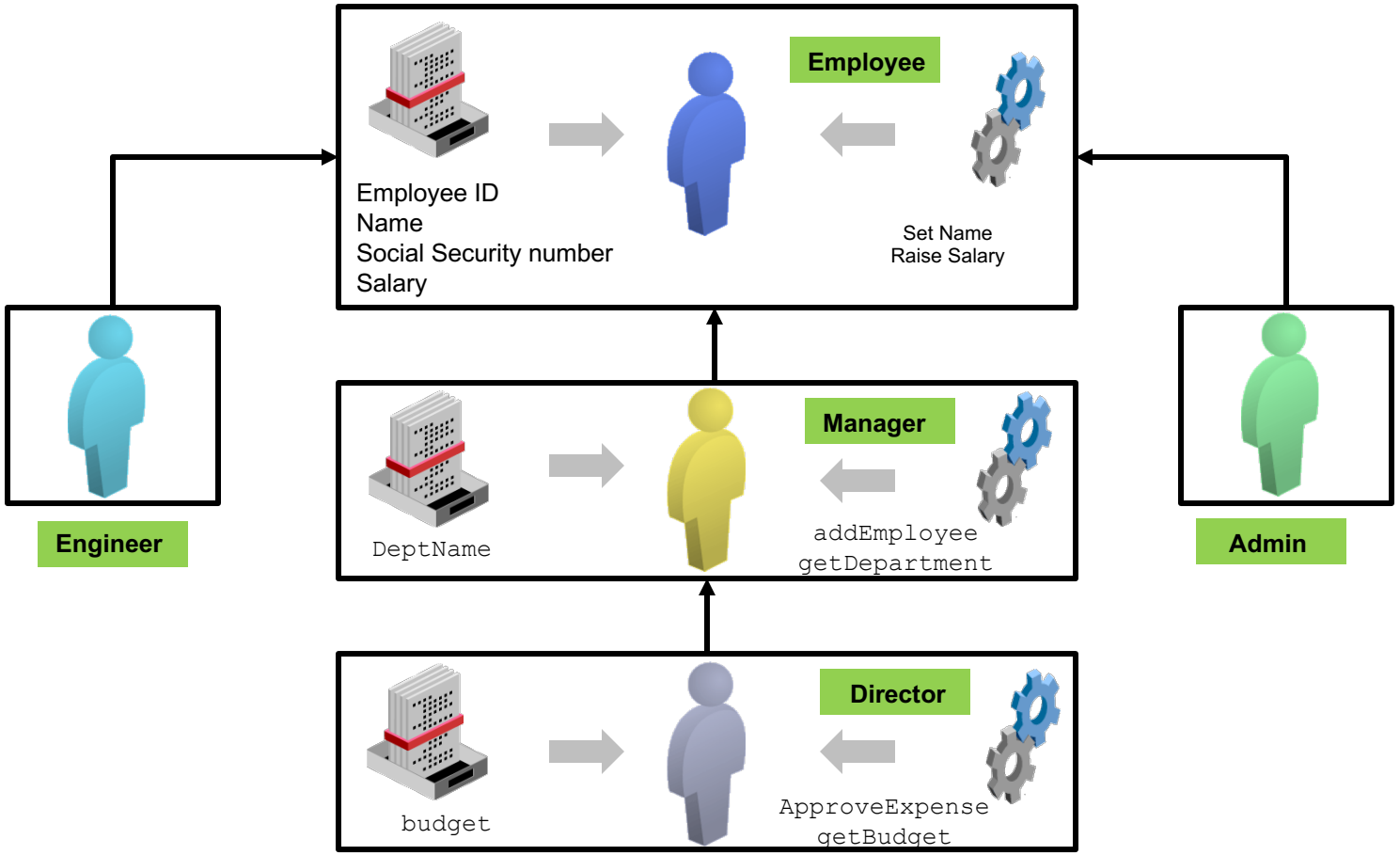
- In addition to overloading methods, you can overload constructors.
- The overloaded constructor is called based upon the parameters specified when the `new` is executed.

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;
    }
}
```

Single Inheritance

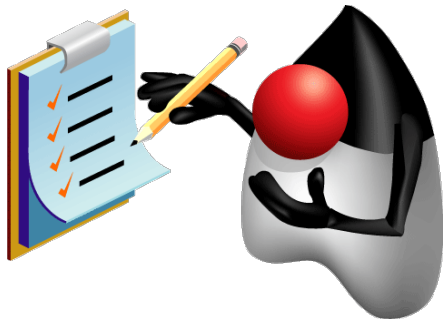
The Java programming language permits a class to extend only one other class. This is called *single inheritance*.



Summary

In this lesson, you should have learned how to:

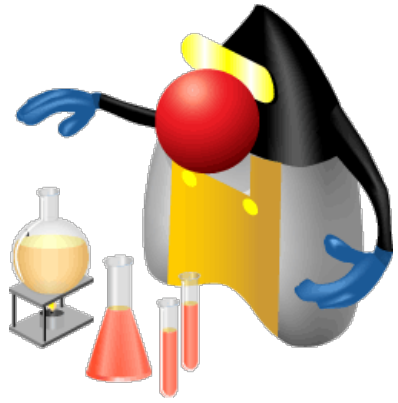
- Use encapsulation in Java class design
- Model business problems by using Java classes
- Make classes immutable
- Create and use Java subclasses
- Overload methods



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.