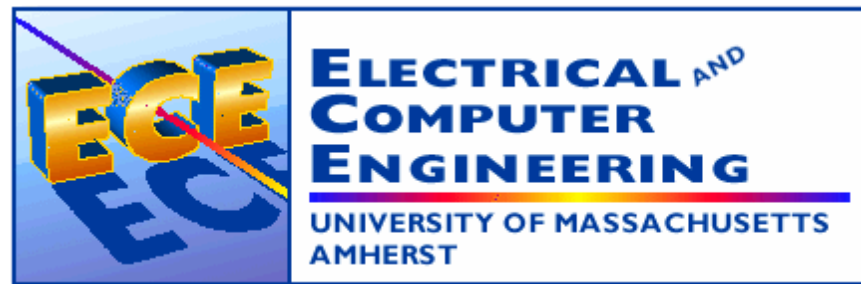

ECE 122

Engineering Problem Solving with Java

Lecture 17

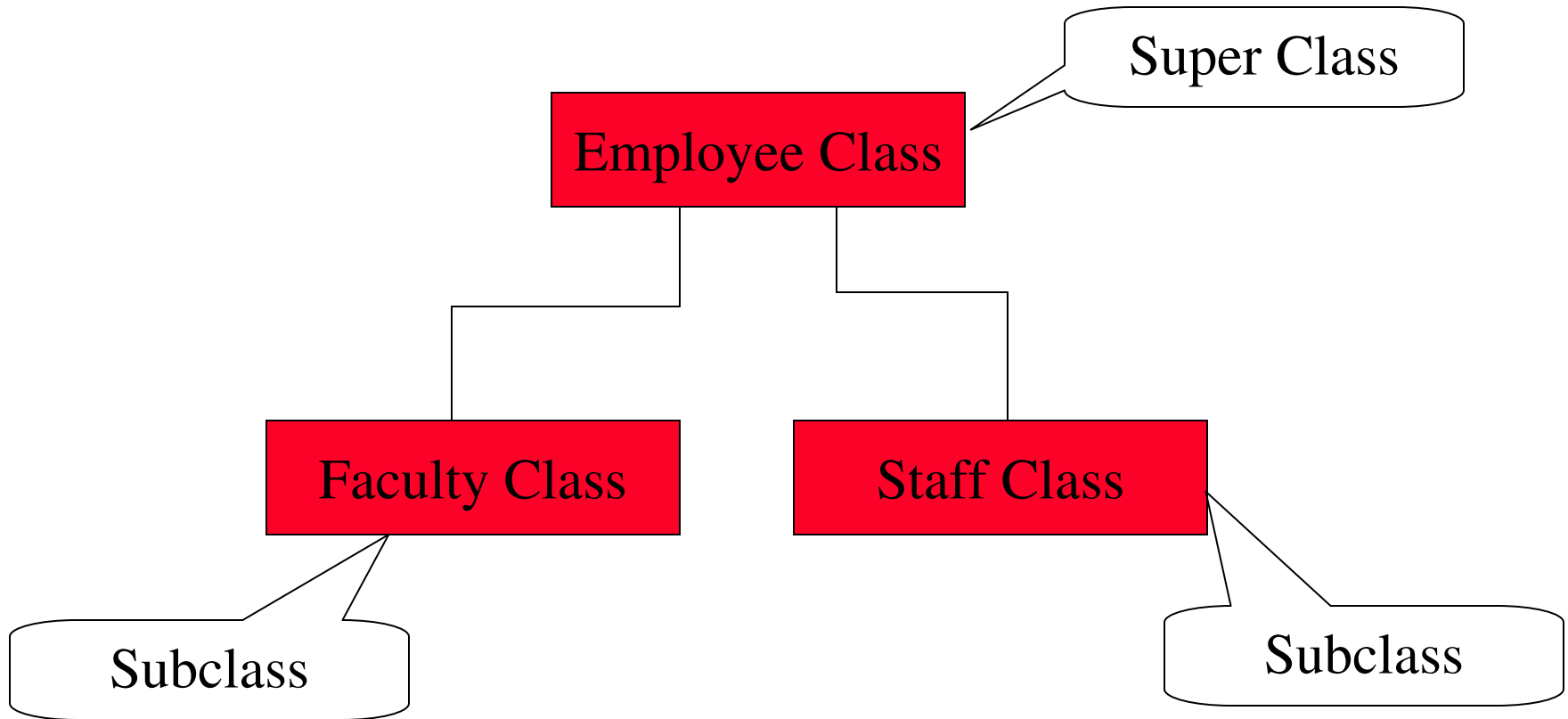
Inheritance



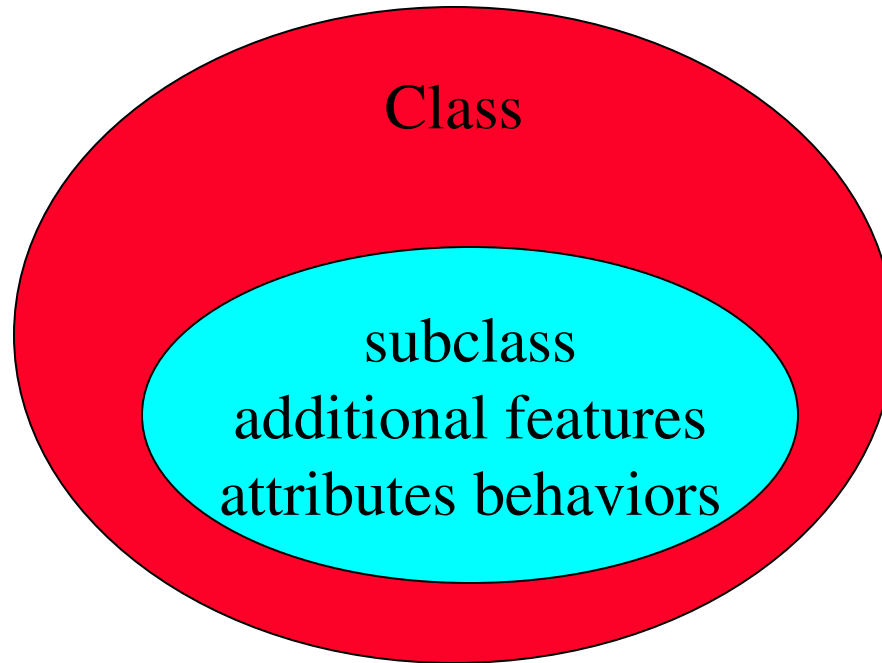
Overview

- **Problem: Can we create bigger classes from smaller ones without having to repeat information?**
- **Subclasses: a class inherits variables and methods from its parents**
- **The leads to code reuse and general code portability**
- **Lots of bookkeeping and notation to keep in mind**

Hierarchy



Subclass



Extends

```
public class subClassName
    extends superClassName
{
    variables;
    methods;
    constructor;
}
```

Inheritance

- ***Inheritance*** allows a software developer to derive a new class from an existing one
- The existing class is called the ***parent class***, or ***superclass***, or ***base class***
- The derived class is called the ***child class*** or ***subclass***
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined by the parent class

Inheritance

- **A programmer can tailor a derived class as needed by adding new variables or methods,**
 - Also can be created by modifying the inherited information
- ***Software reuse* is a fundamental benefit of inheritance**
- **Using existing software components to create new ones**
 - We capitalize on all the effort that went into the design, implementation, and testing of the existing software

Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Car extends Vehicle
{
    // class contents
}
```

The protected Modifier

- **Visibility modifiers affect the way that class members can be used in a child class**
- **Variables and methods declared with private visibility cannot be referenced by name in a child class**
- **They can be referenced in the child class if they are declared with public visibility**
 - **Public variables violate the principle of encapsulation**
- **There is a third visibility modifier that helps in inheritance situations: `protected`**

The protected Modifier

- **The protected modifier allows a child class to reference a variable or method directly in the child class**
- **Provides more encapsulation than public visibility,**
 - **Not as tightly encapsulated as private visibility**
- **A protected variable is visible to any class in the same package as the parent class**

The super Reference

- **Constructors are not inherited, even though they have public visibility**
- **Yet we often want to use the parent's constructor to set up the "parent's part" of the object**
- **The `super` reference can be used to refer to the parent class,**
 - **Often is used to invoke the parent's constructor**

Super

- Call the method in the superclass
- In constructor, super must be in the first line

```
public Faculty(String n, String t, String s) {  
    super(n, t);  
    secName = s;  
}
```

It will call the Employee's constructor:

```
public Employee(String n, String t) {  
    name = n;  
    tel = t;  
}
```

The super Reference

- **A child's constructor is responsible for calling the parent's constructor**
- **The first line of a child's constructor should use the `super` reference to call parent's constructor**
- **The `super` reference is useful**
 - **Can also be used to reference other variables and methods defined in the parent's class**

Staff Class

```
public class Staff extends Employee {  
    private String office;  
    public Staff(String n, String t, String o) {  
        super(n, t); //have to be the first line  
        office = o;  
    }  
    ... .. // you may have other methods here  
}
```

This vs. Super

- **This: the current object**

```
private String name;  
public void setName( String name ) {  
    this.name = name;  
}
```

- **Super: the superclass**

```
public Faculty(String n, String t, String s) {  
    super(n, t); // call superclass Employee(....)  
    secName = s;  
}
```

Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes,
 - Classes inherit the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance

Overriding Methods (not Overloading!)

- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- → The type of the object executing the method determines which version of the method is invoked

Overriding Methods

- **A child class can *override* the definition of an inherited method in favor of its own**
- **The new method must have the same signature as the parent's method, but can have a different body**
- **The type of the object executing the method determines which version of the method is invoked**

Overriding

- A method in the parent class can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Motivation for Overriding

- **Faculty is too busy today to receive a call**
 - make printinfo() method in Faculty's class
 - print the secretary name
 - don't print faculty's telephone
- **Staff helps Faculty to answer the phone**
 - make printinfo() method in Staff's class
 - print name, phone and office address

Overriding Methods

- **In subclass, create a new method with the same name in superclass**
- **Write new code for this method, or extend more function for this method**
- **Example:**
 - **In Faculty class, we will create a totally new printinfo() method**
 - **In Staff class, we will create a new printinfo() method which calls superclass's printinfo()**

Overriding In Faculty

- **Complete Example**

- subclass: [FacultyOverriding.java](#)
- superclass: [EmployeeWithGetName.java](#)

In FacultyOverriding.java

```
public void printinfo() {  
    System.out.println( getname() + "'s secretary name is "+ secName);  
}
```

In EmployeeWithGetName.java

```
public void printinfo() {  
    System.out.println(name+"'s number is "+tel);  
}
```

Result For Faculty

- Call `printinfo()` in [FacultyOverriding.java](#)
`FacultyOverriding tessier =`
`new FacultyOverriding("Tessier", "0160", "Tessier");`
`tessier.printinfo();`
- Result for `printinfo()`;
Tessier's secretary name is Chris

not

Tessier's number is 0160

Overriding In Staff

◦ Complete Example

- subclass: [StaffOverriding.java](#)
- superclass: [EmployeeWithGetName.java](#)

In StaffOverriding.java

```
public void printinfo() {  
    super.printinfo();  
    System.out.println("office address is " + office);  
}
```

In EmployeeWithGetName.java

```
public void printinfo() {  
    System.out.println(name + "'s number is " + tel);  
}
```

Result For Staff

- Call `println()` in [StaffOverriding.java](#)

```
StaffOverriding chris =
```

```
    new StaffOverriding("Chris", "4999", "KEB309");
```

```
chris.println();
```

- Result for `println()`;

Chris' number is 4999

Office address is KEB309

not

Chris's number is 4999

Overloading vs. Overriding

- **Overloading deals with multiple methods with the same name in the same class**
 - **Methods have different signatures**
- **Overriding deals with two methods, one in a parent class and one in a child class,**
 - **They have the same signature**
- **Overloading lets you define a similar operation in different ways for different parameters**
- **Overriding lets you define a similar operation in different ways for different object types**

Summary

- **Inheritance is an important part of object oriented programming**
- **Overriding methods provides a powerful way to customize classes without rewriting the class**
- **Important to understand the implication of public, private, and protected variables**
- **Be sure to review the specific details**
 - **Use of super, use of final, constructor creation, etc**