Chapter 3: Inheritance

Outline

- Create and interact with objects that inherit from other objects
- Create **subclasses**
- Explain **abstract methods** and classes
- Define reusable classes based on inheritance and **abstract classes** and abstract methods.
- Differentiate the abstract classes and Java **interfaces**.
- Define methods, using the **protected** modifier.

Inheritance

- **Inheritance** is an object-oriented mechanism that derives a new class from an existing class.
- The Java programming language allows a class to extend only ***one*** other class. This restriction is referred to as ***single inheritance***.
- **inheritance** allows defining a class in terms of a previously defined class. In the Java programming language, this can be achieved by the keyword **extends**.

- public class Employee {
  public String name = "";
  public double salary;
  public Date birthDate;
  public String getDetails() {...}
  }
- public class Manager {
  public String name = "";
  public double salary;
  public Date birthDate;
  public String department;
  public String getDetails() {...}
  }

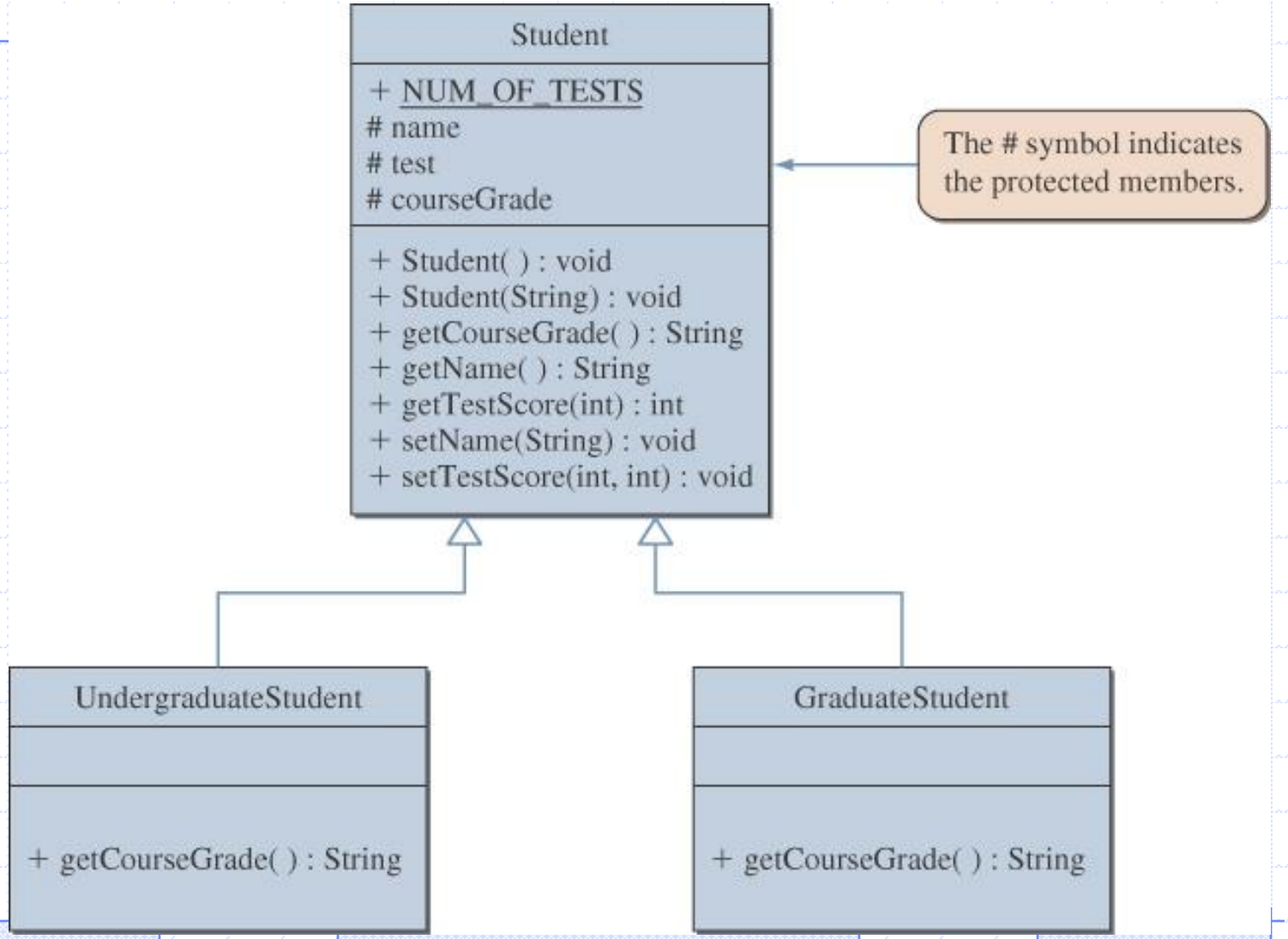
- Manager class is created as a subclass of the Employee class.
- The Manager class is defined to have all the variables and methods that an Employee object has.

```
public class Manager extends Employee {
    public String department;
}
```

Inheritance Relationship

- Inheritance is appropriate where an "is a" relationship exists between two classes.
- one of the classes ***is a*** specialized version of the other.
  - A manager *is an* employee
  - A taxi is a car
  - An UndergraduateStudent is a Student
  - A GraduateStudent is a Student

Inheritance Hierarchy



Identification of What is Inherited

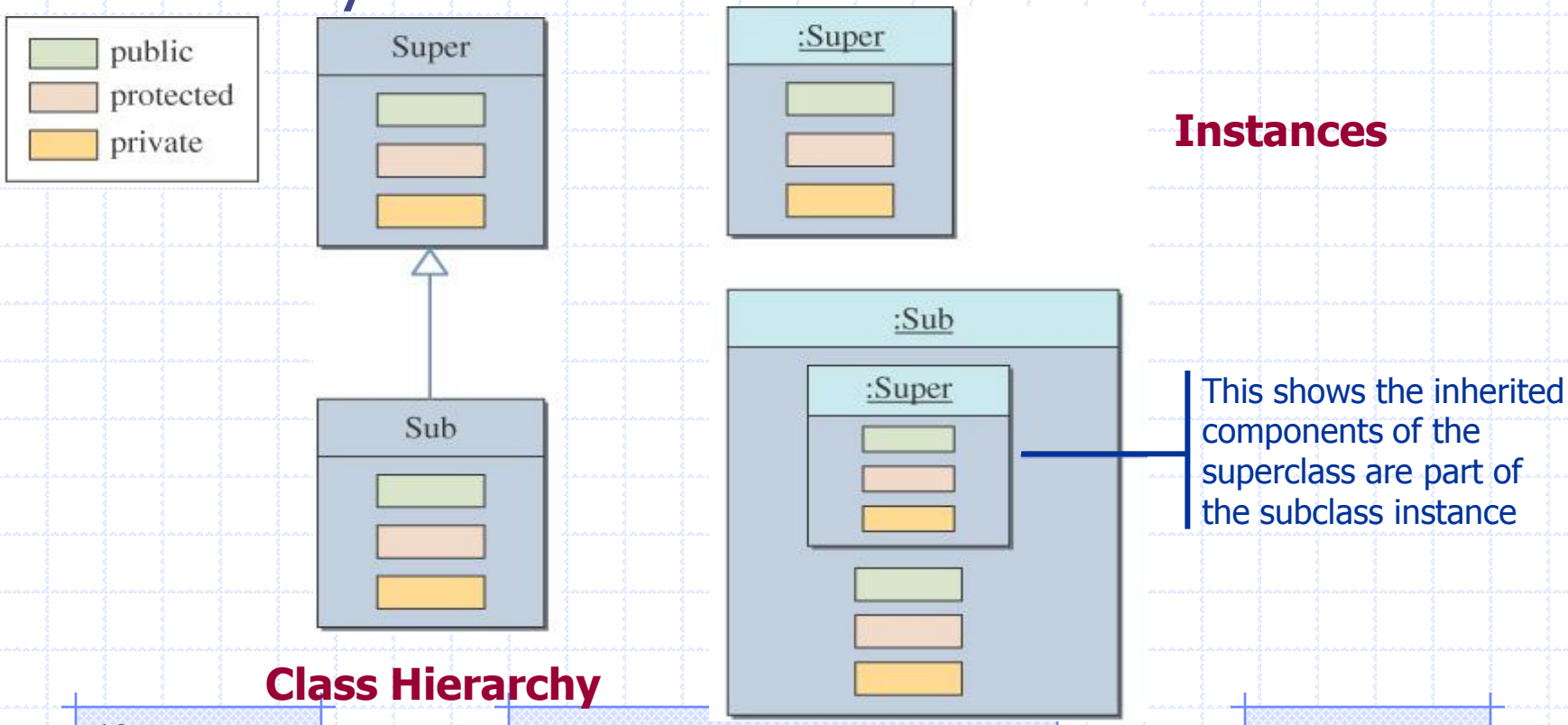
- Before beginning the specialization process, it is essential to determine precisely what will be inherited from the parent class.
- Use the access control rules to identify the attributes and methods that the subclass will inherit from the parent class.
- The key factors that determine which attributes and methods are inherited are the access control modifiers **public, protected, private**, default access, and the packages to which the parent class and the child class belong:

The Protected Modifier

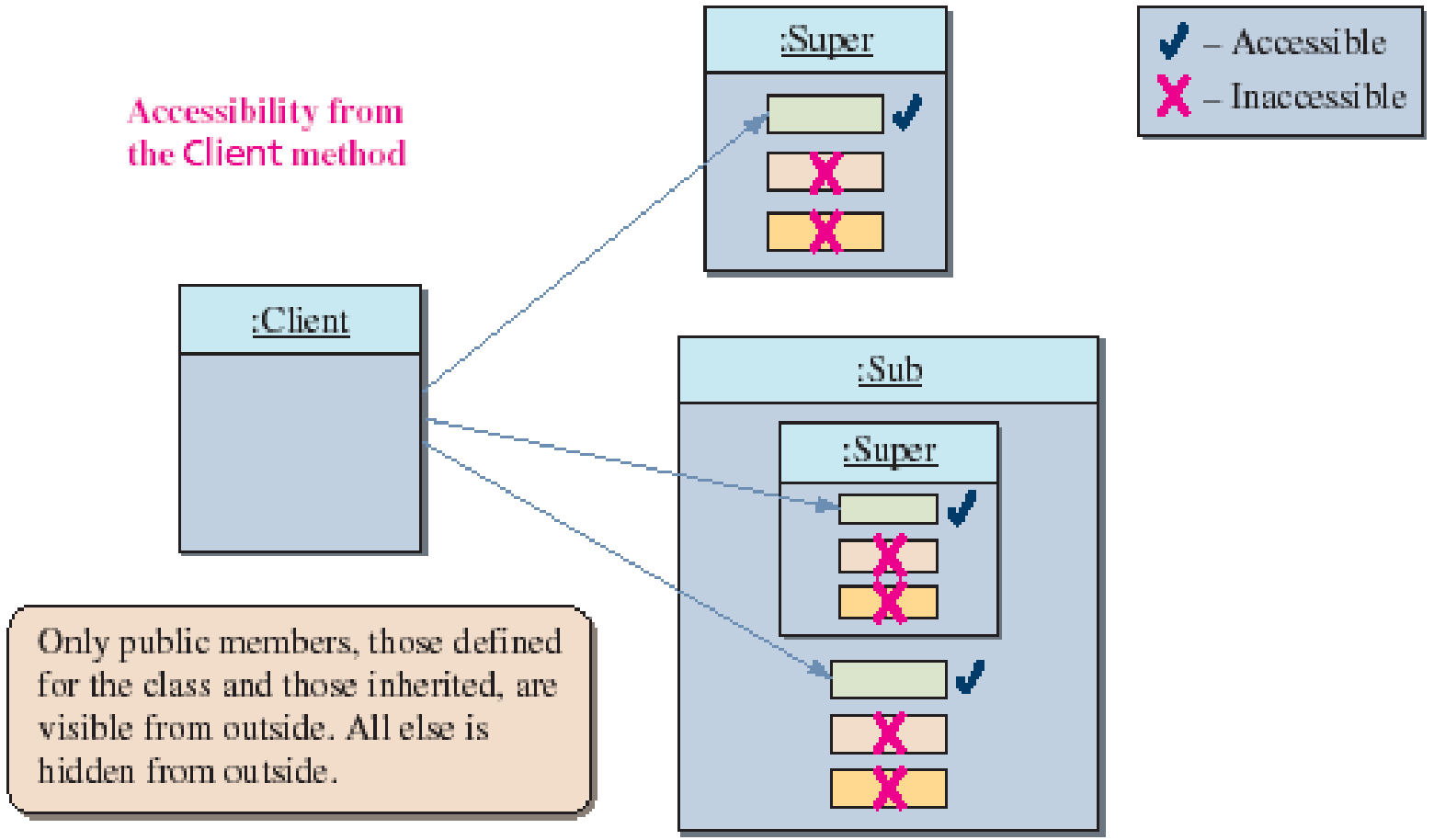
- The modifier **protected** makes a data member or method visible and accessible to the instances of the class and the descendant classes.
- **Public** data members and methods are accessible to everyone.
- **Private** data members and methods are accessible only to instances of the class.

Inheritance and Member Accessibility

- We use the following visual representation of inheritance to illustrate data member accessibility.



The Effect of Three Visibility Modifiers



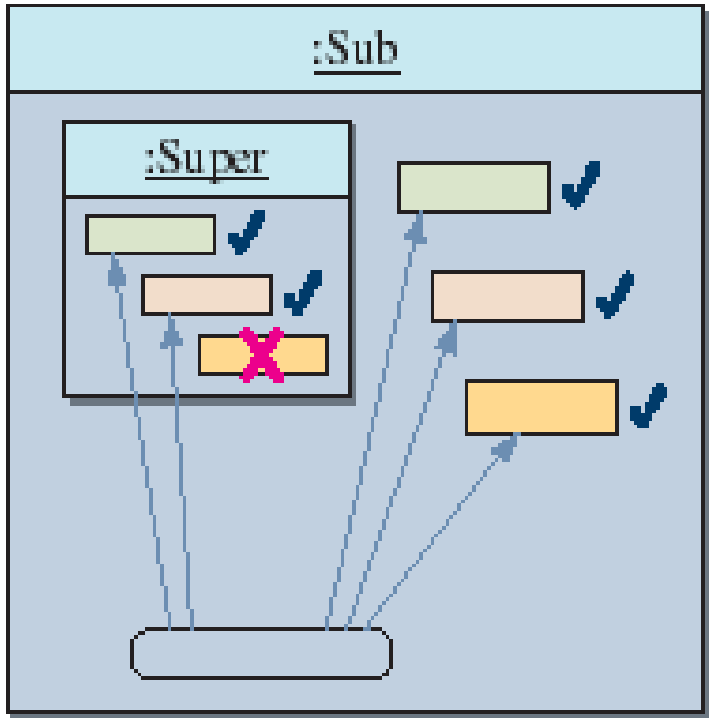
Accessibility of Super from Sub

- Everything except the private members of the Super class is visible from a method of the Sub class.

Accessibility from a method of the Sub class

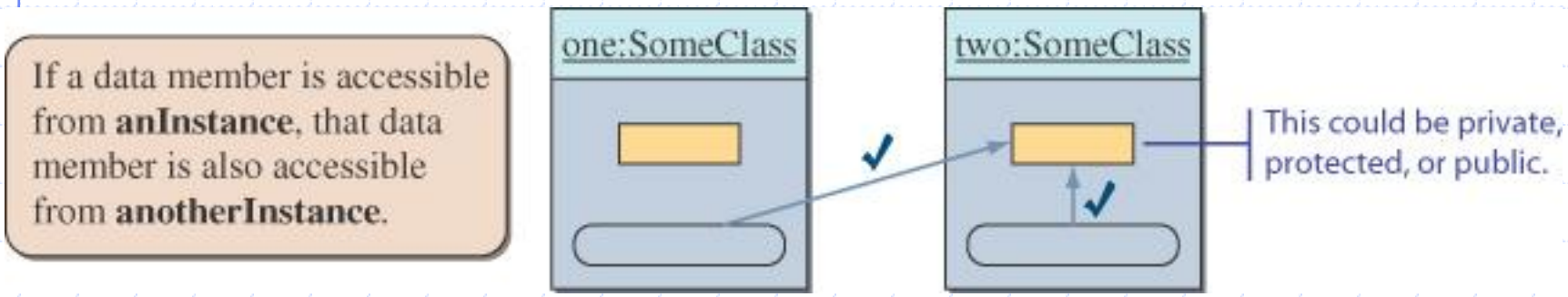
– Accessible
– Inaccessible

From a method of **Sub**, everything is visible except the private members of its superclass.

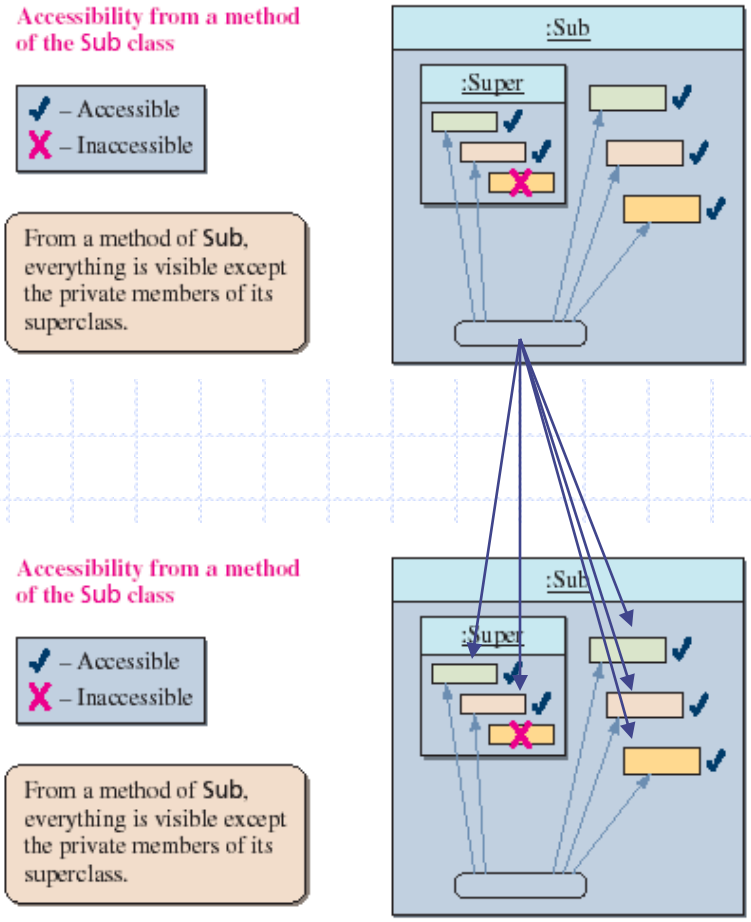


Accessibility from Another Instance

- Data members accessible from an instance are also accessible from other instances of the same class.



Accessibility from Another Instance



Inheritance and Constructors

- Unlike members of a superclass, constructors of a superclass are *not* inherited by its subclasses.
- You must define a constructor for a class or use the default constructor added by the compiler.
- The statement

```
super();
```

calls the superclass's constructor.

Subclass Constructor behavior

- If the subclass constructor does not call the constructor of the superclass, then the default behavior is to always call (implicit call) the constructor with no arguments of the parent class. If the parent class does not define a constructor as having no arguments, then the subclass will not compile
- The invocation of the parent class constructor occurs before the first executable line of the subclass constructor.

Implicit call to the superclass constructor

```
public class Employee {
    String name;
    public Employee() {
            name = "unallocated";
    }
    public Employee(String n) {    name = n;    }
}
public class Manager extends Employee {
    String department;
    public Manager(String s, String d) {
            // Will call no-arg parent constructor Employee()
            // Inherited attribute name is set to "unallocated"
        department = d;
    }
}
```

Implicit call to the superclass constructor

```
public class Employee {
    String name;

public Employee(String n) {    name = n;    }
}
public class Manager extends Employee {
    String department;
    public Manager(String s, String d) {
            // Will call no-arg parent constructor Employee()
            // here, we will get a compile error.

        department = d;
    }
}
```

Implicit call to the superclass constructor

```
public class Employee {
    String name;

}
public class Manager extends Employee {
    String department;
    public Manager(String s, String d) {
            // Will call no-arg parent constructor Employee()
            // here, the default constructor of Employee is called.

        department = d;
    }
}
```

Implicit call to the superclass constructor

```
public class Employee {
    private String name;

}
public class Manager extends Employee {
    String department;
    public Manager(String s, String d) {
            // Will call no-arg parent constructor Employee()
            // here, the default constructor of Employee is called.

            name = s;  // error: name is a private attribute
            department = d;
    }
}
```

Implicit call to the superclass constructor

```
public class Employee {
    protected String name;

    public Employee() {
            name = "unallocated";
    }
    public Employee(String n) {    name = n;    }
}
public class Manager extends Employee {
    String department;
    public Manager(String s, String d) {
            // Will call no-arg parent constructor Employee()
            // Inherited attribute name is set to "unallocated"
            name = s;
            department = d;
    }
}
```

Management of Parent Class Constructor Invocation

- We can **explicitly** invoke a particular parent class constructor as part of a child class initialization by using the keyword **super** from the child constructor's first line.
- The **super** keyword refers to the superclass of the class in which the keyword is used.
- We can use any number of appropriate arguments by using the keyword super to call the appropriate constructor of the parent class.
- **If there is no such parent constructor, a compile error occurs.**

Explicit call to the superclass constructor

```
public class Employee {
        private String name;
        public Employee() {  name = "unallocated";  }
        public Employee(String n) { name = n;}
}
public class Manager extends Employee {
        String department;
        public Manager(String s, String d) {
                super(s);       // Call specific parent constructor
                                // with String argument
                department = d;
        }
}
```

Constructor Chaining

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee’s overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee's no-arg constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

8. Execute println

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println

Overriding Methods

- When a subclass inherits a method, it inherits the interface of the method as well as the implementation of the method from its parent class.
  - For some methods, the parent implementation might not be appropriate for the subclass.
- A method of a new class that has the name, return type, and arguments exactly as a method in the parent class, then the new method is said to override the old one.

Method Overriden

```
public class Employee {
    String name;
    int salary;
    public String getDetails() {
      return "Name: " + name + "\n” +  "Salary: " + salary;
    }
}

public class Manager extends Employee {
    String department;
    public String getDetails() {
      return "Name: " + name + "\n” + "Manager of " + department;
    }
}
```

Invoking Overridden Methods Using the super Keyword

- when we override a method, our goal is not to replace the existing behavior but to extend that behavior in some way.

```
public class Employee {
    private String name;
    private double salary;
    private Date birthDate;
    public String getDetails() {
        return "Name: " + name + "\nSalary: " + salary;
    }
}
public class Manager extends Employee {
    private String department;
    public String getDetails() {    // call parent method
        return super.getDetails() + "\nDepartment: " + department;
    }
}
```

Abstract Methods and Classes

- An ***abstract method*** is a method that is declared but not yet implemented. No body definition is provided. To declare an abstract method:
  - Use the abstract modifier to mark the method as abstract
  - Replace the body of the method with a semicolon (;) character.
- For example:

  public abstract double area();
- An abstract method can only be declared in an abstract class

What Is an Abstract Class?

- An *abstract class* is a class that is incomplete or deemed to be incomplete and is required to be declared by using the abstract modifier.
- A class must use the abstract modifier if any of the following are true:
  - It explicitly contains a declaration of an **abstract method**.
  - It **inherits an abstract method** from its superclass and does not provide an implementation for the method.

Abstract class declaration

```
public abstract class Shape {
    public abstract double area();
    public abstract double perimeter();
    public void display() {
      System.out.println(area()+ "--" +perimeter());
    }
}
```

```
//  the file Rectangle.java
public class Rectangle extends Shape {
    private double l;
    private double w;
    public Rectangle(double x, double y) { l=x; w=y; }
    public final double area() { return (l*w);}
    public double perimeter() { return (2*(l+w)); }
}
```

Declaring an Abstract Class

- Declare the abstract class using the **abstract** keyword.
- Declare the attributes, if any, for the class.
- Declare the abstract methods, if any, in the body of the class.
- Declare the concrete methods, if any, for the class.
- Declare the constructors, if any, for the class.

Creating the Implementing Subclasses

- Declare the subclass as extending the abstract class.
- Override each abstract method of the parent class
  - provide an implementation for each abstract method.
- Override, if necessary, the concrete methods of the parent abstract class.
- If necessary, add new methods and attributes to the subclass for a particular specialization.
- Add any necessary constructors.

Working With Abstract Class Reference Types

- As with a concrete class, an abstract class can be used as a data type for a reference variable declaration.
- The following statement is legal:

  Shape myShape;
- An abstract class cannot be instantiated.
- The following will cause a compile time error if the Account class is an abstract class.

  myShape = new Shape(); // will not compile
- Only concrete classes can be instantiated. Consequently, a reference variable of an abstract class data type must be assigned an instance of a concrete subclass of the abstract class.
- The following compiles if the **Rectangle** class is a concrete subclass of the abstract class ***Shape***.

  myShape = new **Rectangle***();* // will compile

final Modifier

- Final class: The Java allows to apply the keyword final to classes. If this is done, the class cannot be subclassed.
- **final** class MyDate {
  - // this class cannot be subclassed . . .
- }
- Final attribute:
  - all objects will have the same value for the final variable
    - public **static** final int MAX_ARRAY_SIZE = 25;
  - each object has an individual value for the final variable. Any attempt to change the value of a final variable after it has been set causes a compiler error.
    - public final int maxArraySize;
    - myArray(int maxSize) {
      maxArraySize = maxSize;
    - }

Final Methods

- Final method : Methods declared with the final keyword cannot be overridden.

```
public class Rectangle extends Shape {
    private double l;
    private double w;
    public Rectangle(double x, double y) { l=x; w=y; }
    public final double area() { return (l*w);}
    public double perimeter() { return (2*(l+w)); }
}

public class Square extends Rectangle {
    public double area() { return l*l;} // will not compile
    public double perimeter() { return 4*l;} // will compile
}
```