Programming in Java

Class Design

Hua Huang, Ph.D.

Spring 2019

Objectives

- Define **inheritance**, **polymorphism**, **overloading**, **overriding**, and **virtual method invocation**
- Use the **access modifiers** `protected` and the default (package-friendly)
- Describe the concepts of **constructor and method overloading**
- Describe the complete **object construction** and **initialization** operation

Relevance

- How does the Java programming language support object inheritance?

Concept of Inheritance

- The `Employee` class is shown here.

Employee
+**name**: String = "" +**salary**: double +**birthDate**: Date
+getDetails(): String

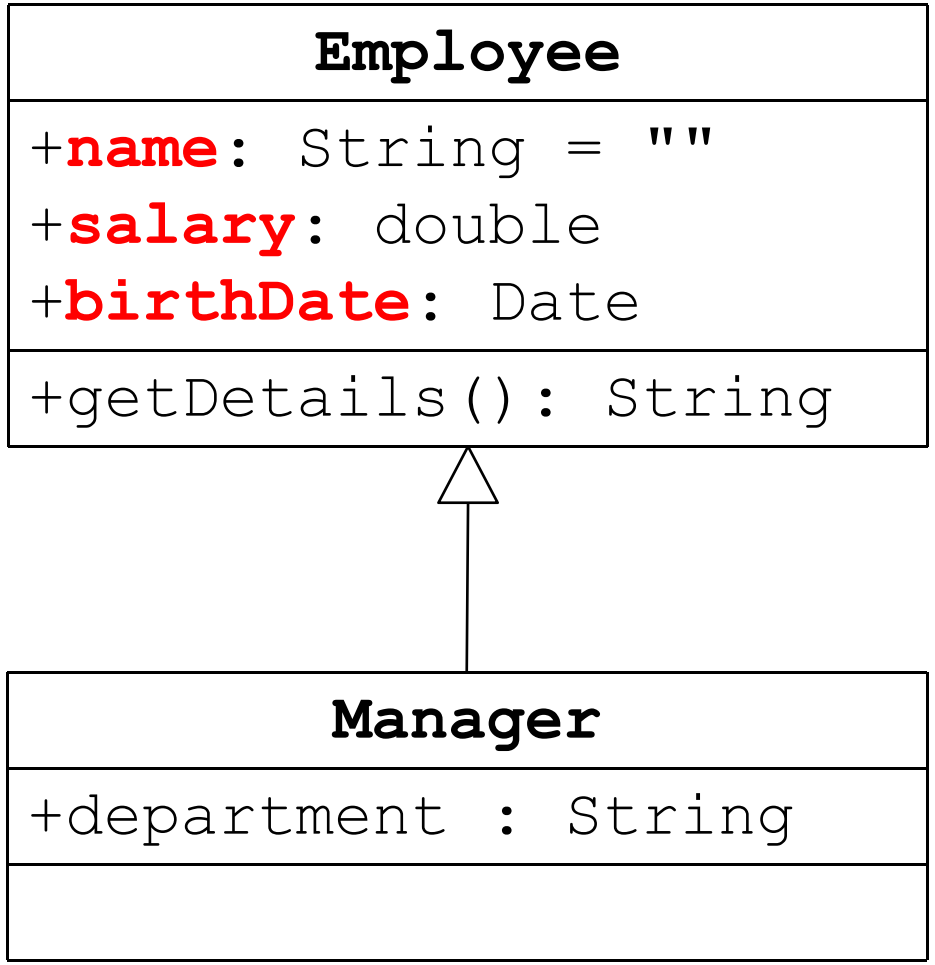
Manager
+**name**: String = "" +**salary**: double +**birthDate**: Date +department : String
+getDetails(): String

```
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() {...}
}
```

Concept of Inheritance(Cont.)

```
1    public class Manager extends Employee {
2        public String department;
3    }
```

- Inheritance provides the following benefits:
  - Enables the creation of **specialized** types
  - **Eliminates duplication**
  - Assists **maintainability**



Implementing Inheritance in Java Technology

1. Select the **parent** class.
2. Examine the parent class to **determine what is inherited** from the parent class.
3. **Declare the subclass**.
4. **Add the attributes and methods** specific to the subclass.
5. If required, **override** the parent class methods.
6. Add **constructors** to the subclass, as needed.

Single Inheritance

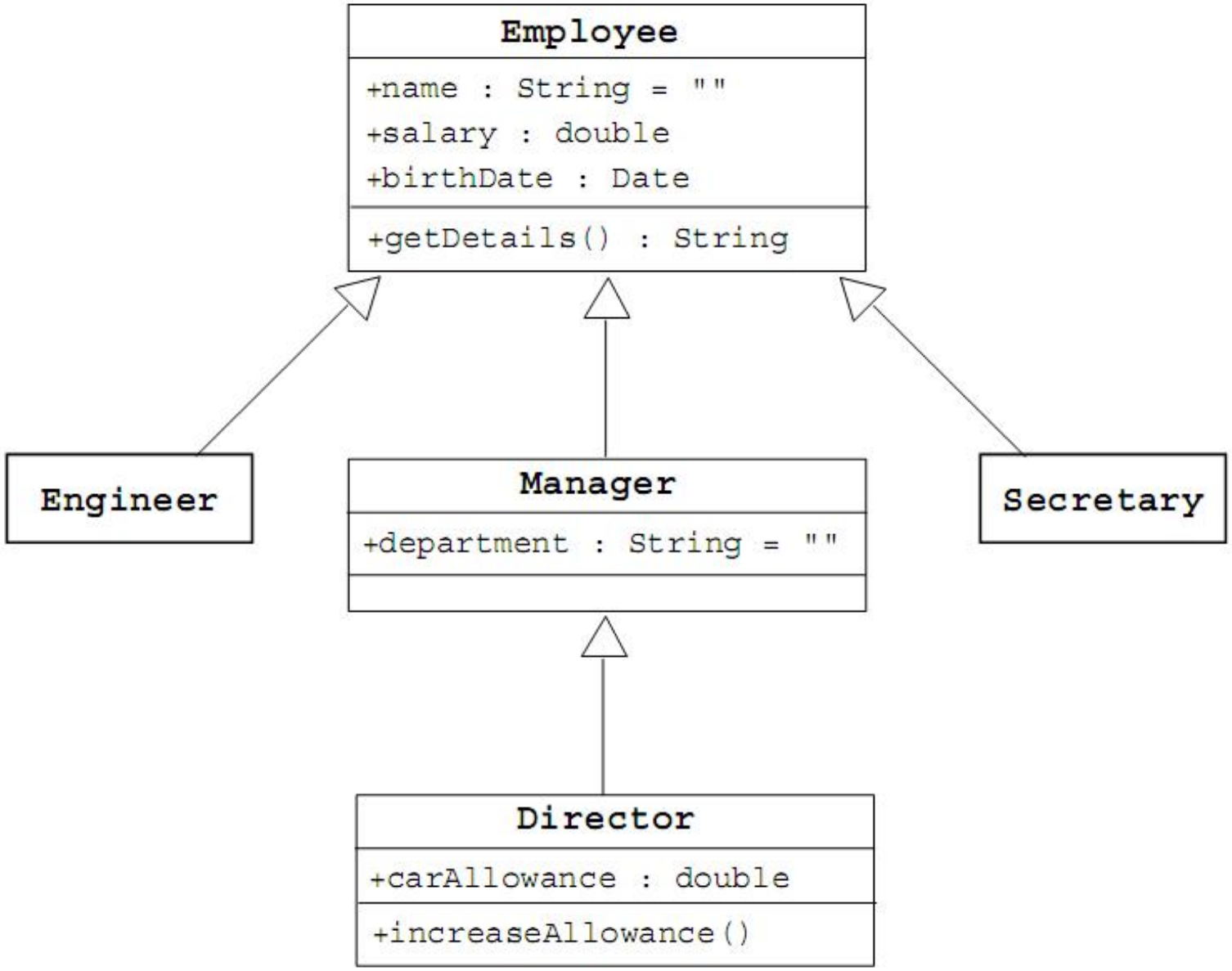
- When a class inherits from only one class, it is called **single inheritance**.

- Interfaces provide the benefits of multiple inheritance without drawbacks.

- Syntax of a Java class is as follows:

```
<modifier> class <name> [extends <superclass>] {
        <declaration>*
}
```

Single Inheritance



Access Control

- Access modifiers on class member declarations are listed here.

Modifier	Same Class	Same Package	Subclass	Universe
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

Overriding Methods

- A **subclass** can modify behavior inherited from a **parent class**.
- A subclass can create a method with different functionality than the parent's method but with the same:
  - **Name**
  - **Return type**[1]
  - **Argument list**

1. Since J2SE version 5, the return type can be a subclass of the overridden return type.

Overriding Methods

```
1     public class Employee {
2       protected String name;
3       protected double salary;
4       protected Date birthDate;
5
6       public String getDetails() {
7         return “Name: “ + name + “\n” + “Salary: “ + salary;
8       }
9     }
```

```
1     public class Manager extends Employee {
2       protected String department;
3
4       public String getDetails() {
5         return “Name: “ + name + “\n” +
6         “Salary: “ + salary + "\n" + “Manager of: “ + department;
7       }
8     }
```

Overridden Methods **Cannot Be Less Accessible**

```
1     public class Parent {
2       public void doSomething() {}
3     }
```

```
1     public class Child extends Parent {
2       private void doSomething() {} // illegal
3     }
```

```
1     public class UseBoth {
2       public void doOtherThing() {
3         Parent p1 = new Parent();
4         Parent p2 = new Child();
5         p1.doSomething();
6         p2.doSomething();
7       }
8     }
```

Invoking Overridden Methods

- A subclass method may **invoke** a **superclass method** using the `super` keyword:
  - The keyword `super` is used in a class to **refer to its superclass**.
  - The keyword `super` is used to **refer to the members of superclass**, both data attributes and methods.
  - Behavior invoked does not have to be in the superclass; it can be further up **in the hierarchy**.

Invoking Overridden Methods(Cont.)

```
1    public class Employee {
2       private String name;
3       private double salary;
4       private Date birthDate;
5
6       public String getDetails() {
7           return "Name: " + name + "\nSalary: " + salary;
8       }
9    }
```

```
1    public class Manager extends    Employee {
2       private String department;
3
4       public String getDetails()     {
5           // call parent method
6           return super.getDetails()
7           + "\nDepartment: " + department;
8       }
9    }
```

Polymorphism

- **Polymorphism** is the **ability to have many different forms**; for example, the `Manager` class has access to methods from `Employee` class.

- An **object has only one form**.
- A **reference variable** **can refer to objects of different forms**.

```
Employee e = new Manager(); // legal

// illegal attempt to assign Manager attribute
e.department = "Sales";
// the variable is declared as an Employee type,
// even though the Manager object has that attribute
```

Virtual Method Invocation

- Virtual method invocation is performed as follows:

```
Employee e = new Manager();
e.getDetails(); // Which method is called ??
```

- **Compile-time type** and **runtime type** invocations have the following characteristics:
  - The **method name** must be **a member of the declared variable type**; in this case `Employee` has a method called `getDetails`.
  - The **method implementation** used is **based on the runtime object's type**; in this case the `Manager` class has an implementation of the `getDetails` method.

Heterogeneous Collections

- Collections of objects with the same class type are called **homogeneous collections**. For example:

```
MyDate[] dates = new MyDate[2];
dates[0] = new MyDate(22, 12, 1964);
dates[1] = new MyDate(22, 7, 1964);
```

- Collections of objects with different class types are called **heterogeneous collections**. For example:

```
Employee[] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
staff[2] = new Engineer();
```

Polymorphic Arguments

- Because a `Manager` is an `Employee`, the following is valid:

```
public class TaxService {
  public TaxRate findTaxRate(Employee e) {
    // calculate the employee's tax rate
  }
}

// Meanwhile, elsewhere in the application class
TaxService taxSvc = new TaxService();
Manager m = new Manager();
TaxRate t = taxSvc.findTaxRate(m);
```

The `instanceof` Operator

```
public class Employee extends Object{}
public class Manager extends Employee{}
public class Engineer extends Employee{}
-------------------------------------
// some other class method
public void doSomething(Employee e) {
    if ( e instanceof Manager ) {
        // Process a Manager
    } else if ( e instanceof Engineer ) {
        // Process an Engineer
    } else {
        // Process any other type of Employee
    }
}
```

Casting Objects

```
public void doSomething(Employee e) {
   if ( e instanceof Manager ) {
      Manager m = (Manager) e;
      System.out.println("This is the manager of "
      + m.getDepartment());
   }
   // rest of operation
}
```

Casting Objects(Cont.)

- Use `instanceof` to **test the type of an object**.
- Restore full functionality of an object by **casting**.
- Check for **proper casting** using the following **guidelines**:
  - Casts **upward** in the hierarchy are done **implicitly**.
  - **Downward casts must be to a subclass and checked by the compiler**.
  - The **object type is checked at runtime** when runtime errors can occur.

Overloading Methods

- Use overloading as follows:

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

- **Argument lists must differ.**
- **Return types** can be different.

Methods Using Variable Arguments

- Methods using **variable arguments** permit multiple number of arguments in methods (valid since JDK 1.5). For example:

```
public class Statistics {
  public float average(int... nums) {
    int sum = 0;
    for ( int x : nums ) {
      sum += x;
    }
    return ((float) sum) / nums.length;
  }
}
```

- The ***vararg*** parameter is treated as an **array**. For example:

```
float gradePointAverage = stats.average(4, 3, 4);
float averageAge = stats.average(24, 32, 27, 18);
```

Overloading Constructors

- As with methods, constructors can be overloaded. An example is:

```
public Employee(String name, double salary, Date DoB)
public Employee(String name, double salary)
public Employee(String name, Date DoB)
```

- **Argument lists** must **differ**.
- You can use the **this** reference **at the first line of a constructor** to call another constructor.

Overloading Constructors(Cont.)

```
1     public class Employee {
2       private static final double BASE_SALARY = 15000.00;
3       private String name;
4       private double salary;
5       private Date birthDate;
6
7       public Employee(String name, double salary, Date DoB) {
8         this.name = name;
9         this.salary = salary;
10        this.birthDate = DoB;
11      }
12      public Employee(String name, double salary) {
13        this(name, salary, null);
14      }
15      public Employee(String name, Date DoB) {
16        this(name, BASE_SALARY, DoB);
17      }
18      // more Employee code...
19    }
```

Constructors Are Not Inherited

- A subclass **inherits all methods and variables** from the superclass (parent class).

Access Modifier	**Inheritance Rules**
private	**Inherited but not accessible**
default	**Inherited** and **accessible** if the subclass and parent class are **in the same package**
protected	Inherited and accessible
public	Inherited and accessible

- A subclass does **not inherit the constructor** from the superclass.
- Two ways to include a constructor are:
  - Use the **default constructor**.
  - Write one or more **explicit constructors**.

Invoking Parent Class Constructors

- To **invoke a parent constructor**, you **must** place a call to `super` in the first line of the constructor.
- You can **call a specific parent constructor** by the **arguments** that you use in the call to `super`.
- If **no** `this` or `super` **call is used in a constructor**, then the **compiler adds an implicit call to `super`()** that calls the parent's no argument constructor (which could be the ***default*** constructor).

  If the **parent class defines constructors**, but does not provide a **no-argument constructor**, then a compiler **error message** is issued.

Invoking Parent Class Constructors(Cont.)

```
1     public class Manager extends Employee {
2       private String department;
3
4       public Manager(String name, double salary, String dept) {
5         super(name, salary);
6         department = dept;
7       }
8       public Manager(String name, String dept) {
9         super(name);
10        department = dept;
11      }
12      public Manager(String dept) {
13        department = dept;
14      }
15      //more Manager code...
16    }
```

// Are there any errors in this scriplet ??

Constructing and Initializing Objects: A Slight Reprise

- **Memory is allocated** and **default initialization** occurs.
  Instance variable initialization uses these steps recursively:
  1. Bind constructor parameters.
  2. If explicit `this()`, call recursively, and then skip to Step 5.
  3. Call recursively the implicit or explicit `super` call, except for **Object**.
  4. Execute the **explicit instance variable initializers**.
  5. Execute the body of the current **constructor**.

Constructor & Initialization Examples

```
1     public class Object {
2       public Object() {}
3     }
```

```
1     public class Employee extends Object {
2       private String name;
3       private double salary = 15000.00;
4       private Date    birthDate;
5
6       public Employee(String n, Date DoB) {
7         // implicit super();
8         name = n;
9         birthDate = DoB;
10      }
11      public Employee(String n) {
12        this(n, null);
13      }
14    }
```

Constructor & Initialization Examples

```
1     public class Manager extends Employee {
2       private String department;
3
4       public Manager(String n, String d) {
5         super(n);
6         department = d;
7       }
8     }
```

Constructor and Initialization Examples(Cont.)

```
0 Basic initialization
  0.1 Allocate memory for the complete Manager object
  0.2 Initialize all instance variables to their default values (0 or null)
1 Call constructor: Manager("Joe Smith", "Sales")
  1.1 Bind constructor parameters: n="Joe Smith", d="Sales"
  1.2 No explicit this() call
  1.3 Call super(n) for Employee(String)
    1.3.1 Bind constructor parameters: n="Joe Smith"
    1.3.2 Call this(n, null) for Employee(String, Date)
      1.3.2.1 Bind constructor parameters: n="Joe Smith", DoB=null
      1.3.2.2 No explicit this() call
      1.3.2.3 Call super() for Object()
        1.3.2.3.1 No binding necessary
        1.3.2.3.2 No this() call
        1.3.2.3.3 No super() call (Object is the root)
        1.3.2.3.4 No explicit variable initialization for Object
        1.3.2.3.5 No method body to call
      1.3.2.4 Initialize explicit Employee variables: salary=15000.00;
      1.3.2.5 Execute body: name="Joe Smith"; date=null;
    1.3.3 - 1.3.4 Steps skipped
    1.3.5 Execute body: No body in Employee(String)
  1.4 No explicit initializers for Manager
  1.5 Execute body: department="Sales"
```

The `Object` Class

- The `Object` class is the **root** of all classes in Java.
- A class declaration with no `extends` clause implies `extends Object`. For example:

```
public class Employee {
   ...
}
```

is equivalent to:

```
public class Employee extends Object {
   ...
}
```

- Two important methods are:
  - `equals`
  - `toString`

The `equals` Method

- The **==** operator determines **if two references are identical** to each other (that is, **refer to the same object**).
- The **`equals`** method determines if objects are **`equal`** but not necessarily **identical**.
- The **`Object`** implementation of the `equals` method uses the **==** operator.
- User classes can **override** the **`equals`** method to implement a domain-specific test for equality.
- Note: You should override the **`hashCode`** method if you override the **`equals`** method.

An `equals` Example

```
1     public class MyDate {
2     private int day;
3     private int month;
4     private int year;
5
6     public MyDate(int day, int month, int year) {
7           this.day  = day;
8           this.month= month;
9           this.year = year;
10    }
```

An `equals` Example(Cont.)

```
11
12          public boolean equals(Object o) {
13              boolean result = false;
14              if ( (o != null) && (o instanceof MyDate) ) {
15                  MyDate d = (MyDate) o;
16                  if ( (day == d.day) && (month == d.month)
17                      && (year == d.year) ) {
18                      result = true;
19                  }
20              }
21              return result;
22          }
23
24          public int hashCode() {
25              return (day ^ month ^ year);
26          }
27      }
```

An `equals` Example(Cont.)

```
1     class TestEquals {
2         public static void main(String[] args) {
3             MyDate    date1 = new MyDate(14, 3, 1976);
4             MyDate    date2 = new MyDate(14, 3, 1976);
5
6             if ( date1 == date2 ) {
7                 System.out.println("date1 is identical to date2");
8             } else {
9                 System.out.println("date1 is not identical to date2");
10            }
11
12            if ( date1.equals(date2) ) {
13                System.out.println("date1 is  equal to date2");
14            } else {
15                System.out.println("date1 is  not equal to date2");
16            }
```

```
17
18              System.out.println("set date2 = date1;");
19              date2 = date1;
20
21              if ( date1 == date2 ) {
22                  System.out.println("date1 is identical to date2");
23              } else {
24                  System.out.println("date1 is not identical to date2");
25              }
26          }
27      }
```

Question: What is the output??

The `toString` Method

- The `toString` method has the following characteristics:
  - This method **converts an object to a String**.
  - Use this method during **string concatenation**.
  - **Override** this method to provide information about a user-defined object in readable format.
  - Use the wrapper class's `toString` static method to convert **primitive** types to a **`String`**.

Questions or Comments?