Inheritance

Part 2 - Polymorphism

Chapter 5, Core Java Volume I and

Chapter 10, Java How to Program, 10th ed.

Contents

- Polymorphism
- Understanding Method Calls
- Dynamic Binding
- Casting
- More on Overriding
- Protected Access
- Final Classes & Methods

Polymorphism

- Type aspects of inheritance (in Java)
  - Every object of the subclass is also an object of its superclass(*is-a* relationship).
    - E.g. a manager is-a an employee
    - Subclass relationship denotes subtype relationship.
  - An object of the subclass can be used wherever the program expects a superclass object. (*the substitution principle*)
    - in assignment statement e.g. Employee e = new Manager(…);
    - in parameter passing .e.g a method foo(Employee e) matches a call foo(manager)
    - in return value

Polymorphism

- Polymorphism : the provision of *a single interface* to entities of different types
  - e.g. overloading (with same function name)
- Subtype Polymorphism (or Inclusion Polymorphism)
  - A reference variable of a class may denote instances of its subclasses (*is-a* relationship)
  - Methods of the superclass can be overrided in the subclasses (*method overriding*)
  - The type of the object to which the variable refers determines the actual method to use (*dynamic binding*)

Polymorphism

- Example: Employee hierarchy

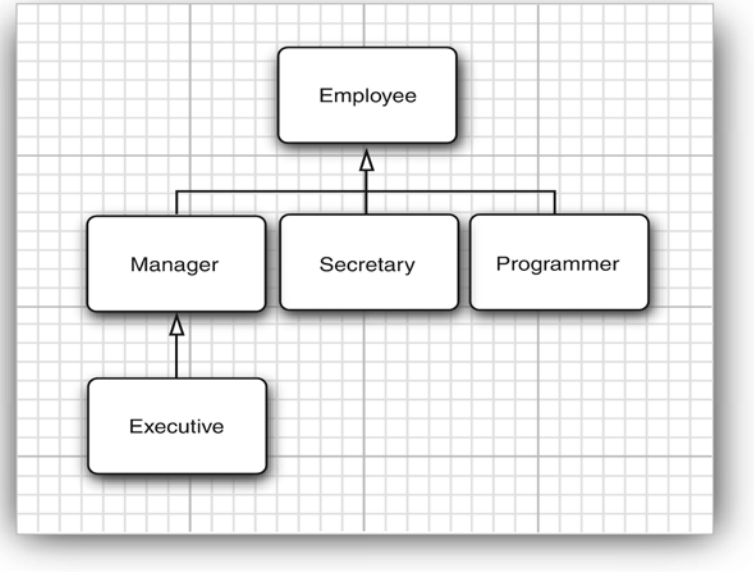
```
Employee[] staff = new Employee[3];
staff[0] = new Employee("Tony Tester", 40000, 1990, 3, 15);
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Manager("Carl Cracker", 80000, 1987, 12, 15);
```

```
for (Employee e : staff)
   System.out.println(e.getName() + " " + e.getSalary());
```

- Which getName method gets called?
  - There is only one: Employee.getName()
- Which getSalary method gets called?
  - Employee.getSalary or Manager.getSalary()?
  - It depends on the actual type of e! (dynamic binding)

Polymorphism

- Advantages of Subtype Polymorphism
  - enables to design and implement systems that are easily extensible
  - enables to write simple and compact programs




```
double totalSalary(Employee[] empList)
{
    double total = 0.0;
    for (Employee e : empList)
        totol += e.getSalary();
    return total;
}
```

Polymorphic!

Compact!
Extensible!

- More useful concepts to provide polymorphism
  - Abstract classes (See Section 5.1.9)
  - Interfaces (See Chapter 6)

Understanding Method Calls

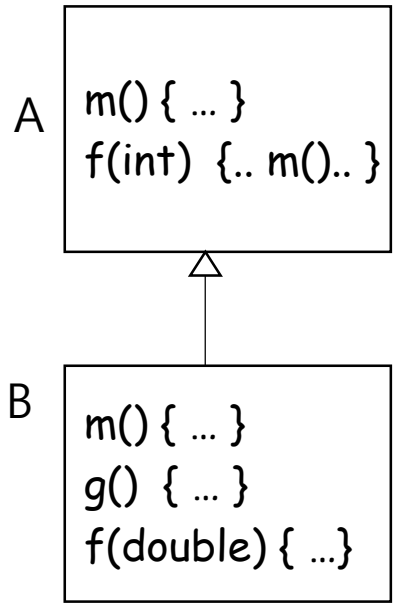
- Suppose `obj` is declared to be of type *C*. Consider a method call:

```
obj.f(args)
```

- Step1: at compile time
  - The compiler finds all accessible methods called `f` in *C* and its superclasses.
  - The compiler selects the method whose parameter types match the argument types (*overloading resolution*).
  - If there is no matching to the call `f`, then comile-error occurs.
  - If the method is private, static, or final, then the compiler binds the matched method to the call `f` (*static binding*).
- Step2: at runtime
  - The exact method is found at runtime (*dynamic binding*).

Understanding Method Calls

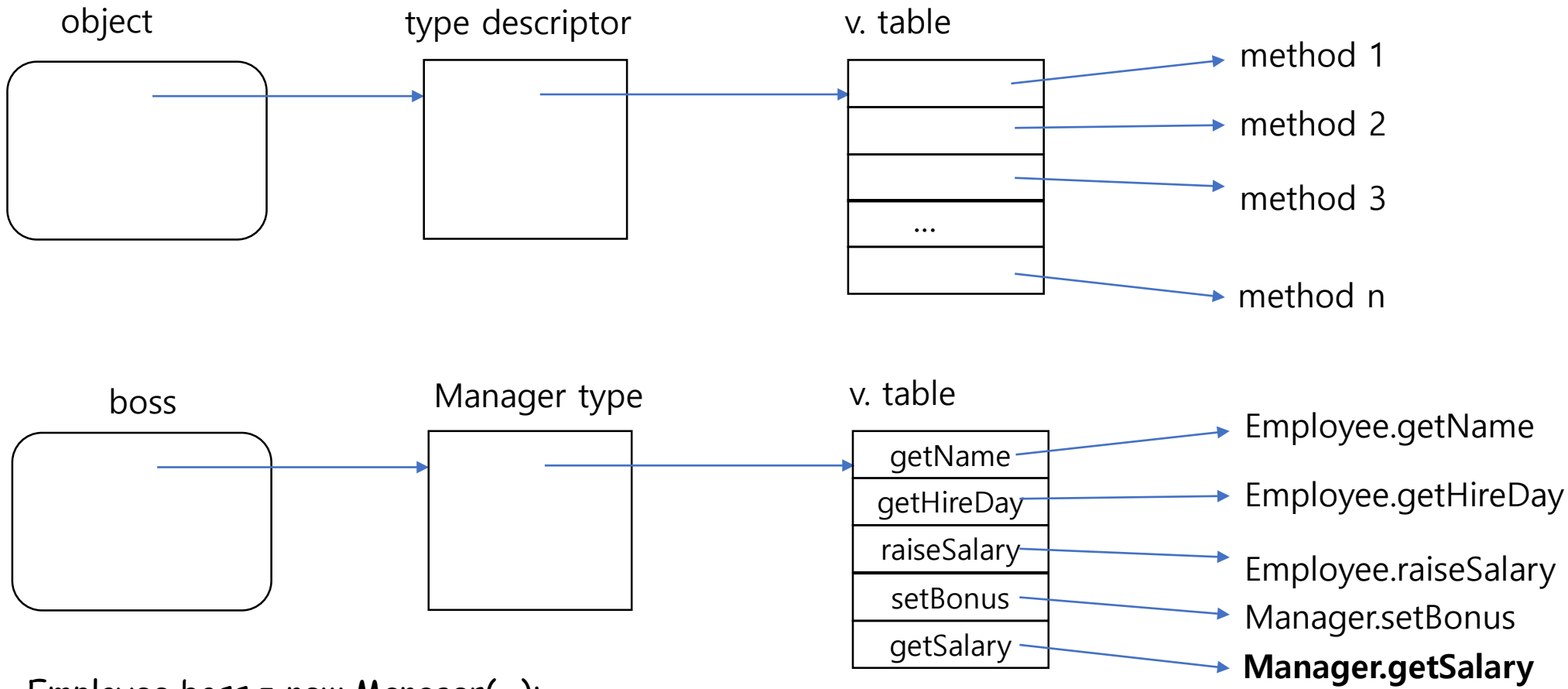
- Example




```
A a = new A();
B b = new B();
A c = b;

a.m();      // m in A
a.f(5);     // f in A  ➔ m in A
a.f(5.5)    // compile error
a.g();      // compile error

b.m();      // m in B
b.f(5);     // f in A  - overloading  ➔ m in B
b.f(5.5)    // f in B  - overloading
b.g();      // g in B

c.m();      // m in B
c.f(10);    // f in A  ➔ m in B
c.g();      // compile error; Why?
```

Implementation of Dynamic Binding

- Virtual Method Table



Casting

- You cannot assign a superclass reference to a subclass variable:

```
Manager m = new Employee(...);     // compile error
Manager boss = staff[2];           // compile error; though staff[2] is an actual Manager.
```

- The Java allows the assignment of a superclass reference to a subclass variable if you *explicitly cast* the superclass reference to the subclass type (*downcasting*)

```
Manager boss = (Manager) staff[2];
boss.setBonus(...);
```

- If staff[0] wasn't actually a Manager, a ClassCastException occurs at runtime.
- Can test with instanceof operator:

```
if (staff[i] instanceof Manager)
{
   boss = (Manager) staff[i];
   . . .
}
```

More about Overriding

- Caution: Argument types of overriding method must match exactly:

```
class Employee
 {
       public void setBoss(Employee boss) {……................}
        .....
 }
class Manager extends Employee
{
      public void setBoss(Manager boss) {.....................} // not overriding, but overloading
      .....
 }
```

- Return type can be *covariant*:

```
class Team
{
  public Employee getBoss() { ... }
  ...
}
```

```
class AvengersTeam extends Team
{
  public Manager getBoss() { ... } // Ok
  ...
}
```

- Use @Override annotation to make the compiler check:

```
@Override
public void setBoss(Manager boss) ..  // in Manager class, compile-error
```

Overriding vs Overloading

- Method overriding and method overloading has common in having the same name in multiple methods within a class or classes in inherited hierarchy
  - method overriding : to provide a more specific(different) implementation in subclass for the method defined in superclass
  - method overloading : to provide multiple different methods with the same name

	Overriding	Overloading
method name	same	same
parameter types	same	different
return type	same or covariant	don't care
binding	dynamic	static
polymorphism	subtype polymorphism	ad-hoc polymorphism
advantage	polymorphism => extensibility	code readability and writability

Protected Access

- A protected field or method is accessible from subclasses:

```
public class Employee
{
   protected double salary;
}
```

  - The salary field can be accessed by **Manager's** methods as well as **Employee's**.

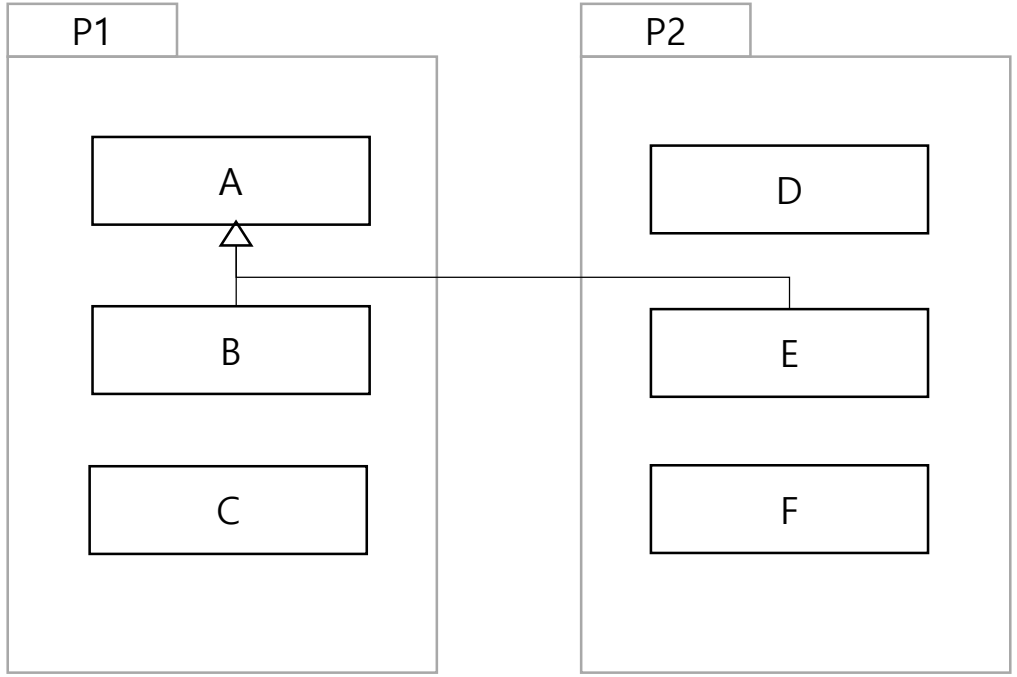
- Caution: Protected features have *package visibility*.
- Caution: Protected fields restrict evolution—anyone can extend a class.
- Protected methods are sometimes useful for methods that are "tricky" to use.
  - protected clone() in **Object** class can be used in all the subclasses (Refer to Section 6.1.9)

Protected Access

- A summary of the four access modifiers in Java that control visibility :

1. **private:** Visible to the class only.
2. **default:** Visible to the package— (no modifiers are needed).
3. **protected:** Visible to the package and all subclasses.
4. **public:** Visible to the every class.

	private	package	protected	public
same class	O	O	O	O
same package		O	O	O
subclasses			O	O
any other classes				O



Final Classes and Methods

- A final class cannot be extended:

```
public final class Executive extends Manager
{
   . . .
}
```

  - There are many final classes in Java API (e.g. String, Math, System, etc.)

- A final method cannot be overridden:

```
public class Employee
{
   . . .
   public final String getName() { return name; }
}
```