# CSE3040 Java Language

## Lecture 11: Object-Oriented Programming (5)

Dept. of Computer Engineering,

Sogang University

## Creating a subclass

- When creating a subclass, its constructor will call the constructor of its superclass.
  - When creating an instance of class Manager, its default constructor is called.
  - The default constructor calls the constructor of its superclass.

```
class Employee {
    private String name;
    private int salary;
    public Employee() {
        name = "NoName";
        salary = 50000;
    }
    public String getName() { return this.name; }
    public int getSalary() { return this.salary; }
}

class Manager extends Employee {
    private int bonus;
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() { return super.getSalary() + this.bonus; }
}
```

```
Manager m = new Manager();          // the default constructor will call the constructor of class Employee
System.out.println(m.getName());    // So the name is initialized to "NoName".
```

## Creating a subclass

- The superclass constructor is called at the beginning of the subclass constructor.

```
class Employee {
    private String name;
    private int salary;
    public Employee() {
        this.name = "NoName";
        this.salary = 50000;
    }
    public void setName(String name) { this.name = name; }
    public String getName() { return this.name; }
    public int getSalary() { return this.salary; }
}
class Manager extends Employee {
    private int bonus;
    public Manager() {
        super.setName("NoName(Manager)");
        bonus = 10000;
    }
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() { return super.getSalary() + this.bonus; }
}
```

```
Manager m = new Manager();         //
System.out.println(m.getName());   // This prints "NoName(Manager)".
System.out.println(m.getSalary()); // We can see that the constructor of Employee is called.
```

## Creating a subclass

- You can explicitly call the superclass constructor using super().
  - The super() statement must be the first statement of the constructor.
    - otherwise the compiler will produce an error.

```
class Employee {
    private String name;
    private int salary;
    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
    public String getName() { return this.name; }
    public int getSalary() { return this.salary; }
}
class Manager extends Employee {
    private int bonus;
    public Manager(String name, int salary) {
        super(name, salary);
        bonus = 10000;
    }
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() { return super.getSalary() + this.bonus; }
}
```

## Creating a subclass

- If you do not explicitly call superclass constructor, it will try to call superclass constructor with no parameter.
  - If superclass constructor with no parameter does not exist, an error will occur.

```
class Employee {
    private String name;
    private int salary;
    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
    public String getName() { return this.name; }
    public int getSalary() { return this.salary; }
}
class Manager extends Employee {
    private int bonus;
    public Manager(String name, int salary) {
        //super(name, salary);       // Error: Employee does not have a constructor with no parameter.
        bonus = 10000;
    }
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() { return super.getSalary() + this.bonus; }
}
```

## Detour: Implicit Constructor

- If a class definition does not contain a constructor, an implicit constructor is assumed to be there.
  - The implicit constructor does not have any parameter, and it does not have any action except calling the superclass constructor (if there is one).

```
class Employee {
    private String name;
    private int salary;
}
```

```
Employee e = new Employee();     // OK
```

- If any constructor is defined, the implicit constructor is no longer used.
  - If a constructor with parameters is defined, then the class does not have a constructor with no parameter.

```
class Employee {
    private String name;
    private int salary;
    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
}
```

```
Employee e = new Employee();     // Error: no Employee() exists
```

## Dynamic method lookup

- We can assign a subclass object to a superclass type variable.

```
Manager boss = new Manager();
Employee empl = boss;
int salary = empl.getSalary();   // call getSalary which is defined in both Employee and Manager.
```

- Then, if we call the method getSalary, the method defined in class Manager is called.
- Although boss is assigned to an Employee type variable, it is still an object of class Manager.
- JVM checks the object's class and calls the right method.
- This is called dynamic method lookup.

- Why do we need to assign a Manager object to an Employee type variable?
  - We can do the following.
  - Manager and Janitor are subclasses of class Employee.

```
Employee[] staff = new Employee[...];
staff[0] = new Employee(...);
staff[1] = new Manager(...);
staff[2] = new Janitor(...);
...
```

## Dynamic method lookup

- Although JVM performs dynamic method lookup at run-time, the following code produces compile error.

```
Employee empl = new Manager("Donald", 100000);
empl.setBonus(20000);
```

- Because the compiler looks for method setBonus in the class Employee when processing the statement:
  - empl.setBonus(20000);

- In order to satisfy the compiler, we need to cast the class to the subclass.

```
Employee empl = new Manager("Donald", 100000);
if(empl instanceof Manager) {
    Manager mgr = (Manager) empl;
    mgr.setBonus(20000);
}
System.out.println(empl.getSalary());
```

## Final method

- You can declare your method as a final method. Then, no subclass can override this method.

```
class Employee {
    private String name;
    private int salary;
    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
    public final String getName() { return this.name; }  // subclass cannot override this method.
    public int getSalary() { return this.salary; }
}
class Manager extends Employee {
    private int bonus;
    public Manager(String name, int salary) {
        super(name, salary);
        bonus = 10000;
    }
    public void setBonus(int bonus) { this.bonus = bonus; }
    public String getName() { return "Sir " + super.getName(); }  // Error: overriding a final method
}
```

- getClass method of class Object is an example of final method.

## Final class

- You can even make your class final. Then, no subclass can inherit from this class.

```
final class Employee {
    private String name;
    private int salary;
    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
    public String getName() { return this.name; }
    public int getSalary() { return this.salary; }
}
class Manager extends Employee {
    ...     // Error: cannot inherit a final class
}
```

- String is an example of final class.

## Abstract method and abstract class

- In a class definition, it is possible to declare a method without an implementation.
- It is called an abstract method, and you need to use keyword abstract at the method header.
- If a class definition includes an abstract method, the class is an abstract class. You should use the keyword abstract at the class too.

```
abstract class Person {
    private String name;

    public Person(String name) { this.name = name; }
    public final String getName() { return name; }

    public abstract int getId();
}
```

- A subclass inheriting class Person must:
  - implement method getId() or,
  - be an abstract class itself
- abstract class vs. interface
  - An abstract class may have instance variables and constructors.
    - An interface cannot have them.

## Abstract class

- You cannot create an instance of an abstract class.

```
Person p = new Person("Fred");  // Error: cannot create instances of an abstract class.
```

- You can create an instance of a subclass inheriting the abstract class and implementing all abstract methods.

```
class Student extends Person {
    private int id;
    public Student(String name, int id) { super(name); this.id = id; }
    public int getId() { return id; }
}
```

```
Student s = new Student("Fred", 1729);
System.out.println(s.getName() + " " + s.getId());
```

- You can also assign a Student object to a Person variable.

```
Person p = new Student("Fred", 1729);
System.out.println(p.getName() + " " + p.getId());
```

## Protected access

- If you declare a variable or a method using access modifier protected, the variable or the method can be accessed from subclasses.

```
class Employee {
    private String name;
    protected int salary;
    public Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
    public String getName() { return this.name; }
    public int getSalary() { return this.salary; }
}
class Manager extends Employee {
    private int bonus;
    public Manager(String name, int salary) {
        super(name, salary);
        bonus = 10000;
    }
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() { return salary + bonus; }  // can access superclass protected variable
}
```

- Protected variables can also be accessed from other classes in the same package
  - Access from non-subclasses of other packages prohibited.
  - Subclasses of other packages are allowed to access protected variables (and methods).

## Superclass and Interface

- A class can implement an interface.

```
class Student implements Named {
    ...
}
```

- A class can implement multiple interfaces.

```
class Student implements Named, Registered {
    ...
}
```

- A class can extend a superclass.

```
class Student extends Person {
    ...
}
```

- A class cannot extend multiple classes.

```
class Student extends Person, Animal {    // Error: cannot extend multiple classes.
    ...
}
```

## Superclass and Interface

- A class can extend a superclass and also implement interfaces.

```
interface Named {
    default String getName() { return "NoName"; }
}
abstract class Person {
    private String name;
    public Person(String name) { this.name = name; }
    public String getName() { return name; }
    public abstract int getId();
}
class Student extends Person implements Named {
    private int id;
    public Student(String name, int id) { super(name); this.id = id; }
    public int getId() { return id; }
}
```

- Notice how method getName() exists in both interface Named and class Person.
  - In this case, the superclass has priority over interfaces.
  - When method getName is called, the method in the superclass is called.

```
Student s = new Student("Fred", 1729);
System.out.println(s.getName());
```

## class Object: superclass of all classes

- In Java, all classes implicitly inherit from class Object.
  - If your class doesn't extend any class, it is implicitly extending class Object.
  - The following two are the same.

```
class Student {
    ...
}
```

```
class Student extends Object {
    ...
}
```

## class Object: superclass of all classes

- Some methods defined in class Object (not a full list)
  - String toString()
    - returns a string representation of an object (e.g. "java.lang.Object@3c407114").
  - boolean equals(Object other)
    - returns true if the object is the same as **other**.
    - returns false if the object is not the same as **other**, or **other** is null.
  - int hashCode()
    - returns the hash code assigned to an object.
      - If two objects are the same, they have the same hash code.
  - Class<?> getClass()
    - returns a Class object that represents the class of the object.
  - protected Object clone()
    - returns a copy of the object
  - protected void finalize()
    - method called when the garbage collector retrieves the object.

## class Object: method toString

- Returns a string representation of the object.

```
class Employee {
    private String name;
    protected int salary;
    public Employee(String name, int salary) { this.name = name; this.salary = salary; }
    public String getName() { return this.name; }
    public int getSalary() { return this.salary; }
}
```

```
Employee empl = new Employee("John", 50000);
System.out.println(empl.getName() + " " + empl.getSalary());
System.out.println(empl.toString());
```

- If you just try to print the object, it automatically calls the toString method of the object.

```
Employee empl = new Employee("John", 50000);
System.out.println(empl.getName() + " " + empl.getSalary());
System.out.println(empl);    // this will print empl.toString()
```

# class Object: method toString

- You can override this method in your class.

```
class Employee {
    private String name;
    protected int salary;
    public Employee(String name, int salary) { this.name = name; this.salary = salary; }
    public String toString() { return getClass().getName() + "[name=" + name + ",salary=" + salary + "]"; }
    public String getName() { return this.name; }
    public int getSalary() { return this.salary; }
}
```

- If your superclass has a toString method, you can call that class in the subclass.

```
class Manager extends Employee {
    private int bonus;
    public Manager(String name, int salary) { super(name, salary); bonus = 10000; }
    public String toString() { return super.toString() + "[bonus=" + bonus + "]"; }
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() { return salary + bonus; }
}
```

```
Employee empl = new Manager("Peter", 100000);
System.out.println(empl);
```

## Programming Lab #11

## 11-01. Constructors for Inherited Classes

- Run the following code and understand the result.

```
class Employee {
    private String name;
    private int salary;
    public Employee() {
        name = "NoName";
        salary = 50000;
    }
    public void setName(String name) { this.name = name; }
    public String getName() { return this.name; }
    public int getSalary() { return this.salary; }
}

class Manager extends Employee {
    private int bonus;
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() { return super.getSalary() + this.bonus; }
}

public class Ex11_01 {
    public static void main(String[] args) {
        Manager m = new Manager();
        System.out.println(m.getName());
        System.out.println(m.getSalary());
    }
}
```

# 11-01. Constructors for Inherited Classes

- Try the following modifications to the code in the previous slide.
  - Add an explicit constructor for class Manager that sets the default name to "NoName(Manager)" and sets the bonus to 10000.
  - Change the constructor of class Employee so that it takes two arguments, name and salary, and sets the instance variables.
    - This will cause a compile error.
  - Change the constructor of class Manager so that the error is addressed.
  - In the main method, declare a variable of class Employee and assign a Manager object to the variable. Call getSalary() of the object and see the result. (dynamic method lookup)
  - Try setting the bonus of the object by calling instance method setBonus. This will produce an error. Fix the error so that setBonus is successfully called.

## 11-02. Abstract Classes

- class Person is an abstract class. Implement class Student so that the main method can run successfully without error.

```
abstract class Person {
    private String name;
    public Person(String name) { this.name = name; }
    public final String getName() { return name; }
    public abstract int getId();
}
```

```
public class Ex11_02 {
    public static void main(String[] args) {
        Person p = new Student("Fred", 1729);
        System.out.println(p.getName() + " " + p.getId());
    }
}
```

```
Fred 1729
```

## 11-03. toString()

- Modify the class definitions so that the main method produces results shown below.

```
class Employee {
    private String name;
    protected int salary;
    public Employee(String name, int salary) { this.name = name; this.salary = salary; }
    public String getName() { return this.name; }
    public int getSalary() { return this.salary; }
}
class Manager extends Employee {
    private int bonus;
    public Manager(String name, int salary) { super(name, salary); bonus = 10000; }
    public void setBonus(int bonus) { this.bonus = bonus; }
    public int getSalary() { return salary + bonus; }
}
public class Ex11_03 {
    public static void main(String[] args) {
        Employee e1 = new Employee("John", 50000);
        Manager e2 = new Manager("Peter", 100000);
        System.out.println(e1);
        System.out.println(e2);
    }
}
```

```
John, Employee, salary: 50000
Peter, Manager, salary: 100000, bonus: 10000
```

# End of Class

Instructor office: AS818A

Email: jso1@sogang.ac.kr