Inheritance

Objectives and Outline

- Objectives:
  - Understand java inheritance and learn to use it.
- Outline
  - Introduction: concept of inheritance
  - Deriving a subclass
  - Using subclasses
  - Special class types and classes arising from inheritance
    - Abstract classes
    - Final classes
    - The Object class
    - (The Class class allows you to analyze and manipulate java program at run time.)

Introduction to Inheritance

- Technique for deriving a new class from an existing class.

- Existing class called **superclass, base class, or parent class.**

- New class is called **subclass, derived class, or child class.**

Introduction to Inheritance

- Subclass and superclass are closely related
    - Subclass share fields and methods of superclass
    - Subclass can have more fields and methods
    - Implementations of a method in superclass and subclass can be different
    - An object of subclass is automatically an object of superclass, but not vice versa
        - The set of subclass objects is a subset of the set of superclass objects. (E.g. The set of Managers is a subset of the set of Employees.) This explains the term subclass and superclass.

Introduction to Inheritance

Why inheritance?

- Employee class:
  name, salary, hireDay;
  getName, raiseSalary(), getHireDay().

- Manager **is-a** Employee, has all above, and
  - Has a bonus
  - getsalary() computed differently

- Instead of defining Manager class from scratch, one can derive it from the Employee class. Work saved.

Introduction to Inheritance

Why inheritance?

Inheritance allows one to factor out common functionality by moving it to a superclass, results in better program.

Checking

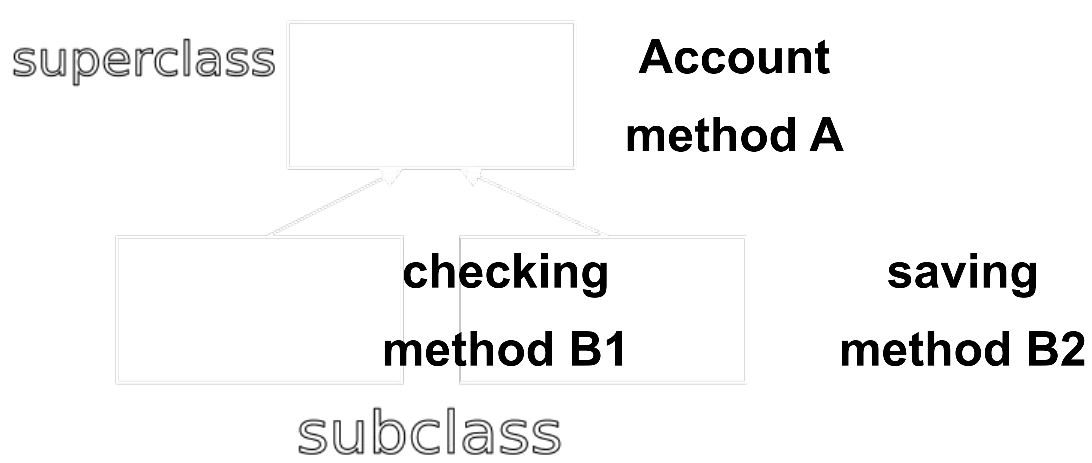
method A

method B1

Saving

method A

method B2



Introduction to Inheritance

- Multiple inheritance
  - A class extends >1 superclasses
- Java does not support multiple inheritance
  - A java class can only extend ONE superclass
  - Functionality of multiple inheritance recovered by interfaces.

Outline

- Outline
  - Introduction: concept of inheritance
  - Deriving a subclass
  - Using subclasses
  - Special class types and classes arising from inheritance
    - Abstract classes
    - Final classes
    - The Object class

Deriving a Subclass

- General scheme for deriving a subclass:

```
class subClassName extends superClassName
{
constructors

refined methods
additional methods

additional fields
}
```

Indicate the differences between subclass and superclass

```
class Employee
{
public Employee(String n, double s, int year, int month, int day) {…}
public String getName(){…}
public double getSalary() {…}
public Data getHireDay(){…}
public void raiseSalary(double byPercent) {…}
private String name;
private double Salary;
private Date hireDay;
}
```

Deriving a class

- Extending Employee class to get Manager class

```
class Manager extends Employee
{ public Manager(...) {...} // constructor
public void getSalary(...) {...} // refined method
// additional methods
public void setBonus(double b){...}

// additional field
private double bonus;
}
```

Deriving a Class

- Plan
  - Fields of subclass
  - Constructors of subclass
  - Methods of subclass
  - A few notes

Fields of subclass

- Semantically: Fields of superclass + additional fields
  - Employee
    - Name, salary, hireday
  - Manager
    - name, salary, hireday
    - bonus

- Methods in subclass cannot access private fields of superclass.
  - After all, subclass is another class viewed from super class.
  - More on this later.

Fields of subclass

- Static instance fields are inherited but not duplicated in subclass.

```
class Employee //StaticInherit.java
{ public Employee (...)
{ ...
numCreated++;
}
public static int getNumCreated()
{ return numCreated; }
...
private static int numCreated=0;
}
Manager b = new Manager(...); // numCreated = 1
```

Fields of subclass

- To count number of Managers separately, declare a new static variable in Manager class

```
class Manager extends Employee
{ public Manager (...)
{ ...
numManager++; }
public static int getNumCreated()
{ return numManager; }
...
private static int numManager=0;
}
```

Deriving a Class

- Plan
  - Fields of subclass
  - Constructors of subclass
  - Methods of subclass
  - A few notes

Constructors of Subclass

- Every constructor of a subclass must, directly or indirectly, invoke a constructor of its superclass to initialize fields of the superclass. (Subclass cannot access them directly)

- Use keyword **super** to invoke constructor of the superclass .

```
public Manager(String n, double s, int year, int month, int
day)
{
super(n, s, year, month, day);
bonus = 0;
}
```

Must be the first line

Constructors of Subclass

- Can call another constructor of subclass.
  - Make sure that constructor of superclass is eventually called.

```
public Manager(String n)
{
this(n, 0.0, 0, 0, 0);
}
```

Constructor of Subclass

- If subclass constructor does not call a superclass constructor explicitly, then superclass uses its default constructor.

```
class FirstFrame extends JFrame
{ public FirstFrame()
{
setTitle("FirstFrame");
setSize(300, 200);
}
}
```

super() implicitly called here

Constructors of Subclass

- Constructors are not inherited.
  - Let’s say Employee has two constructors

    public Employee(String n, double s, int year,

    int month, int day)

    public Employee(String n, double s)

  - Manager has one constructor

    public Manager(String n, double s, int year,

    int month, int day)

    new Manager(“George”, 20000, 2001, 7, 20); //ok

    new Manager(“Jin”, 25); //not ok

Deriving a Class

- Plan
  - Fields of subclass
  - Constructors of subclass
  - Methods of subclass
  - A few notes

Methods of Subclass

- Methods of subclass include
  - Non-private methods of superclass that are not refined (inherited).
  - + Refined (overriding) methods
  - + Additional methods
- Refined and additional methods appear in subclass

Overriding Methods

- Salary computation for managers are different from employees. So, we need to modify the getSalary, or provide a new method that overrides getSalary

```
public double getSalary( )
{ double baseSalary = super.getSalary();
return basesalary + bonus;
}
```

Call method of superclass

- Cannot replace the last line with

```
salary += bonus;
```

Because **salary** is private to Employee.

- Cannot drop “super.”, or else we get an infinite loop

Overriding Methods

- An overriding method must have the same **signature** (name and parameter list) as the original method. Otherwise, it is simply a new method:

- Original Method in Employee:
  public double getSalary( ){...}
  public void raiseSalary(double byPercent){...}

- New rather than overriding methods in Manager:
  public void raiseSalary(int byPercent){...}

Overriding Methods

- An overriding method must have the same return type as the original method:
  - The following method definition in Manager would lead to compiler error:

    public int getSalary( ){...}

- An overriding method must be at least as visible as the superclass method.

- private methods cannot be overridden, but others (public, protected, default-access methods) can.

Additional Methods

```
public void setBonus(double b)
{
bonus = b;
}
```

Methods for Subclass Manager

- Inherited from Employee

getName, getHirDay, raiseSalary

- Refined from Employee

getSalary.

- Additional

Manager, setBonus.

Deriving a Class

- Plan
  - Fields of subclass
  - Constructors of subclass
  - Methods of subclass
  - A few notes

Note about this

- Refer to another constructor in the same class

```
class Employee
{
public Employee(String n, double s, int year,
int month, int day){…}
public Employee(String n) // another constructor
{
this(n, 0, 0,0,0); // salary default at 0
}
```

Note about this

- Refers to the current object

```
  public Employee(String n, double s, int year,
  int month, int day)
{
this.name = n;
this.salary = s;
GregorianCalendar calendar
= new GregorianCalendar(year, month - 1, day);
// GregorianCalendar uses 0 for January
this.hireDay = calendar.getTime();
}
```

Note about super

A special keyword that directs the compiler to invoke the superclass constructor and method

- Refers to constructor of superclass

```
public manager(String n, double s, int year,
int month, int day)
{ super(n, s, year, month, day);}
```

- Invokes a superclass method

```
public double getSalary()
{
double baseSalary = super.getSalary();
return baseSalary + bonus;
```

Note about Protected Access

- A subclass can access **protected** fields and methods of a superclass

- Example: If the **hireDay** field of Employee is made protected, then methods of Manager can access it directly.

- However, methods of Manager can only access the **hireDay** field of **Manager** objects, not of other **Employee** objects. (See next slide for more explanation)

- Protected fields are rarely used. Protected methods are more common, e.g. **clone** of the **Object** class (to be discussed later)

Note about Protected Access

```
public class Employee
{ …
protected Date hireDay;
}

Public class Manager extends Employee
{
someMethod()
{
Employee boss = new Manager();
boss.hireDay //ok
Employee clerk = new Employee();
clerk.hireDay //not ok
```

Note about Protected Access

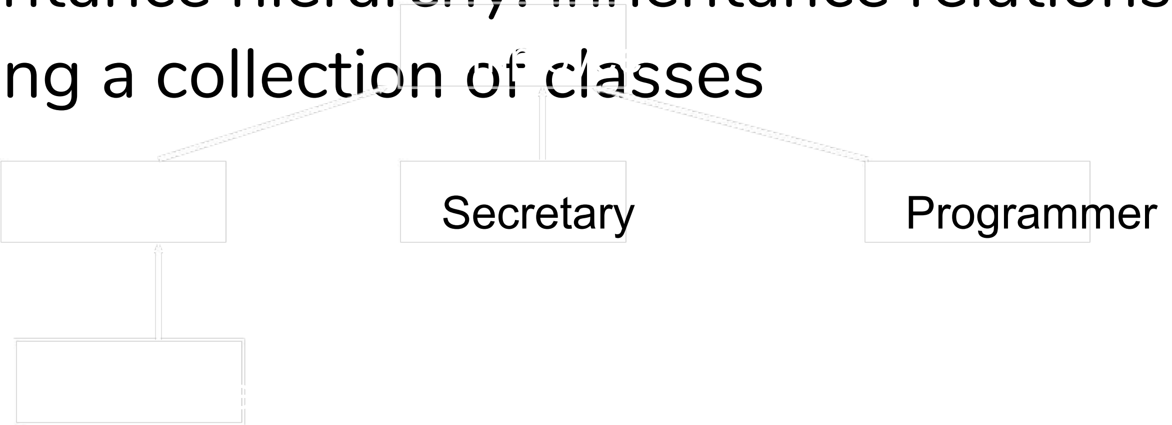
- When to use the protected modifier:
  - Best reserved specifically for subclass use.
  - Do not declare anything as protected unless you know that a subclass absolutely needs it.
    - **clone** of the **Object** class
  - In general, do not declare methods and attributes as protected in the chance that a subclass may need it sometime in the future.
  - If your design does not justify it explicitly, declare everything that is not in the public interface as private.

Note about Inheritance Hierarchies

- Can have multiple layers of inheritance:

  class Executive extends Manager { ....}

- Inheritance hierarchy: inheritance relationships among a collection of classes



Outline

- Outline
  - Introduction: concept of inheritance
  - Deriving a subclass
  - Using subclasses
  - Special class types and classes arising from inheritance
    - Abstract classes
    - Final classes
    - The Object class

Using Subclasses

- Plan:
    - Class compatibility:
        - An object of subclass is automatically an object of superclass, but not vice versa.
            - Employee harry = new Employee();
            - Employee jack = new Manager();
    - Polymorphism:
        - Object variable an refer to multiple actual types
    - Dynamic binding
        - Java’s ability to call the appropriate method depending on actual type of object
            - harry.getSalary();

Class Compatibility

- Object of a subclass can be used in place of an object of a superclass

Manager harry = new Manager(...);

Employee staff = harry;

Employee staff1 = new Manager(...);

harry automatically cast into an Employee, widening casting.

- Why does staff.getSalary() work correctly?
  - Employee has method getSalary. No compiling error.
  - Correct method found at run time via dynamic binding

Class Compatibility

- The opposite is not true

```
Employee harry = new Employee(...);
Manager staff = harry; // compiler error
Manager staff1 = new Employee(...); // compiler error
```

Narrowing cast

- Only necessary when
  - Object of subclass was cast into object of a superclass, and want to get back the original class type

```
        Manager carl = new Manager(...);
Employee staff = carl;
// This will produce a compiler error, since Employee
// doesn’t has setbonus method
staff.setbonus(5000);
//cast is required to get back the original class type
Manager b = (Manager) staff;
b.setbonus(5000);
```

Narrowing cast

- One bad narrowing cast terminates program
  - Make sure narrowing cast is legal using operator instranceof

```
        Manager carl = new Manager(...);
Employee staff1 = carl;
Employee staff2 = new Employee(...);
Manager b1 = (Manager) staff1; // ok
Manager b2 = (Manager) staff2; // crashes
if ( staff2 instanceof Manager )
Manager b2 = (Manager) staff2; // does not crash
```

Using Subclasses

- Plan:
  - Class compatibility:
  - Polymorphism:
  - Dynamic binding

Polymorphism & Dynamic binding

- Method call: case 1

  ```
  Employee harry = new Employee(...);
  harry.getSalary(); // calls method of Employee
  ```

- Method call: case 2

  ```
  Manager carl = new Manager(...);
  carl.getSalary(); // calls method of Manager
  ```

- Method call: case 3

  ```
  Manager carl = new Manager(...);
  Employee staff = carl;
  staff.getSalary();
  ```

  - Calls method of Employee or Manager?
  - Answer: method of Manager.

Polymorphism & Dynamic binding

- How does java call the correct method?
  - Consider method call x.f(args), where x declared as “C x”
    - Compiler
      - Enumerate all methods called f in C and non-private methods named f in superclasses of C.
      - Does overloading resolution.
      - If f is a private, static, final method, compiler knows exactly which method to call. (Static binding).
    - JVM
      - Start with the actual type of x. If f not found there, move to superclass. And so on. (Dynamic binding)
      - Method table utilized to make this more efficient.

Polymorphism & Dynamic binding

```
Manager carl = new Manager(…);
Employee staff = carl;

Staff.getSalary();
//Try Manager. Found.

staff.getHireDay();
//Try Manager. Not found. Move to Employee. Found.
```

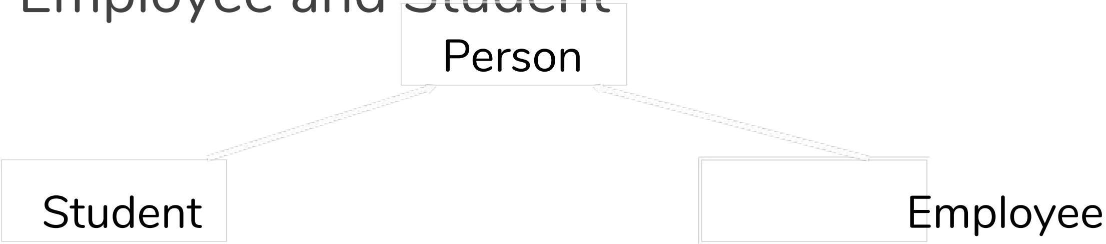
- Implication: method in subclass hides (overrides) method in superclass

Outline

- Outline
  - Introduction: concept of inheritance
  - Deriving a subclass
  - Using subclasses
  - Special class types and classes arising from inheritance
    - Abstract classes
    - Final classes
    - The Object class

Abstract Classes

- Consider two classes: Employee and Student



- Common methods:
  - getName
  - getDescription: returns
    - Employee: “an employee with salary $50,000”
    - Student: “a student majoring in Computer Science”

Abstract Classes

```
class Person
        { public Person(String n) { name = n;}
        public String getName()
        { return name;}

        public String getDescription();
        // but how to write this?

        private String na
```

Abstract Classes

- Solution: leave the **getDescription** method abstract
  - Hence leave the **Person** class abstract

```
abstract class Person
        { public Person(String n)
        { name = n;}

        public abstract String getDescription();

        public String getName()
        { return name;}
        private String name;
        }
```

Abstract Classes

- An abstract method is a method that
  - Cannot be specified in the current class (C++: pure virtual function).
  - Must be implemented in non-abstract subclasses.

- An abstract class is a class that may contain one or more abstract methods

- Notes:
  - An abstract class does not necessarily have abstract method
  - Subclass of a non-abstract class can be abstract.

Abstract Classes

- Cannot create objects of an abstract class:

```
New Person(“Micky Mouse”) // illegal
```

- An abstract class must be extended before use.

```
class Student extends Person
{ public Student(String n, String m)
{ super(n); major = m;}

public String getDescription()
{ return "a student majoring in " + major; }
private String major;
}
```

Abstract Classes

```
class Employee extends Person
{ ...
public String getDescription()
{
NumberFormat formatter
= NumberFormat.getCurrencyInstance();
return "an employee with a salary of "
+ formatter.format(salary);
}
...
private double salary;
}
```

Abstract Classes

```
Person[] people = new Person[2];
people[0]= new Employee("Harry Hacker", 50000, 1989,10,1);
people[1]= new Student("Maria Morris", "computer science");

for (int i = 0; i < people.length; i++)
{ Person p = people[i];
System.out.println(p.getName() + ", "
+ p.getDescription());
} //PersonTest.java
```

- This would not compile if we don’t have getDescription in Person.

Outline

- Outline
  - Introduction: concept of inheritance
  - Deriving a subclass
  - Using subclasses
  - Special class types and classes arising from inheritance
    - Abstract classes
    - Final classes
    - The Object class

Final Methods and Classes

- Final method:
  - Declared with keyword **final**
  - Cannot be overridden in subclasses

```
class Employee

{ ...

public final String getName() {...}

}
```

Final Methods and Classes

- Final class
  - Declared with keyword final
  - Cannot be sub-classed. Opposite of abstract class.
  - All methods are final.

final class Executive extends Manager

{ .... }

Final Methods and Classes

- Reasons to use final methods (and final classes):
  - Efficiency:
    - Compiler put final method in line: e.getName() replaced by e.name.
    - No function call.
    - No dynamic binding.
  - Safety:
    - Other programmers who extend your class cannot redefine a final method.

Final Methods and Classes

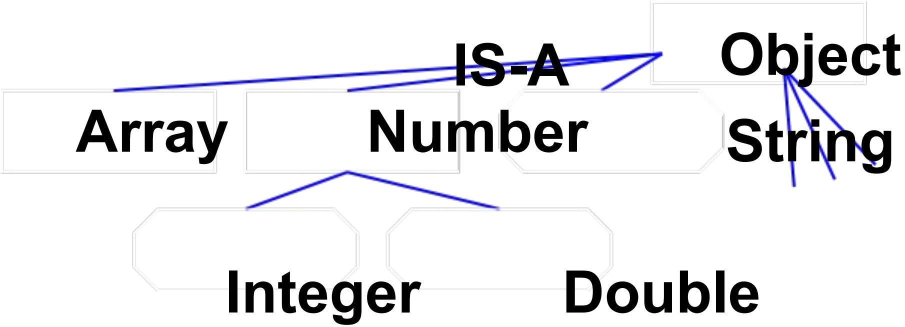
- Wrapper classes for primitive types are final
  - Primitive types
    - boolean, char, byte, short, int, long, float, double, void
  - Wrapper classes
    - Boolean, Character, Byte, Short, Integer, Long, Float, Double, Void
  - Methods of java.lang.Integer
    - int intValue(),
    - Static String toString(int i) // cannot override to print in another way
    - static int parseInt(String s)
    - static Integer valueOf(String s)
    - ...

Outline

- Outline
    - Introduction: concept of inheritance
    - Deriving a subclass
    - Using subclasses
    - Special class types and classes arising from inheritance
        - Abstract classes
        - Final classes
        - The Object class

The Object class

- **The** Object class (java.lang.Object) is the mother of all classes
    - Everything eventually is related to Object



- Never write class Employee extends Object {...} because Object is taken for granted if no explicitly superclass:

    class Employee {...}

The Object class

- Has a small set of methods
  - **boolean equals(Object other);**
    - **x.equals(y);**

      for any reference values x and y, this method returns

      true if and only if x and y refer to the same object (x==y has the value true).
    - Must be overridden for other equality test, e.g. name, or id

      E.g. Overridden in String
  - **String toString**();
    - Returns a string representation of the object
    - **System.out.println(x)** calls **x.toString()**
    - So does **“”+x**
    - Override it if you want better format.
    - Useful for debugging support
  - Other methods to be discussed later.

The Object class

- The Object class gives us one way to do generic programming (There are other ways, e.g. interfaces.)

```
int find(Object[]a, Object key)
{ int i;
for (i=0; i<a.length; i++)
{ if (a[i].equals(key)) return i;
}
return –1; // not found
}
```

- The function can be applied to objects of any class provided that the **equals** method is properly defined for that class

```
Employee[] staff = new Employee[10];
Employee harry;
...
int n = find(staff, harry);
```

Assuming Employee has method

public boolean

The Object class

- Suppose Employee has

  public equals(int ID){...};

  - Can we call find(staff, 5)?
  - No, because 5 is not an object of any class.

- What should we if we want to use ID-based equality test?
  - Use rapper class
    - Instead of **equals(int ID),** define **equals(Integer ID)**
    - Then, call **find(staff, new Integer(5));**

The Object class

- Example of generic programming: java.util.ArrayList
    - Array-like class that manages objects of type Object, and that
    - Grows and shrinks dynamically (no need to specify size)
    - Very useful.

- Methods: //ArrayListTest.java

ArrayList(),ArrayList(int initialCapacity) //constructors

int size()

boolean add(Object obj) // append element

boolean add(int index, Object obj)

void remove(int index)

void set(int index, Object obj)

Object get(int index)

// needs to cast to appropriate type

Summary of Modifiers

- Class Modifiers
    - public: Visible from other packages
    - default (no modifier): Visible in package
    - final: No subclasses
    - abstract: No instances, only subclasses
  - Field modifiers
    - public: visible anywhere
    - protected: visible in package and subclasses
    - default (no modifier): visible in package
    - private: visible only inside class
    - final: Constant
    - static: Class variable

Summary of Modifiers

- Method Modifiers
  - final: No overriding
  - static: Class method
  - abstract: Implemented in subclass
  - native: Implemented in C
  - private, public, protected, default: Like variables