Inheritance

Object-oriented programming
Outline

- What is inheritance?
- Inheritance in Java
- Reuse

Readings:
- *Java how to program*, chapter 9
"is-a" relationship

- Similar things (sharing same set of attributes/operations)
  → a group / a concept

- Similar groups (sharing a subset of attributes/operations)
  → a bigger group / a more general concept
  - A student is a person
  - An employee is a person
  - A manager is an employee

→ An object of the subgroup "is-a" object of the supergroup
Inheritance

- Based on "is-a" relationship
- Objects of subclass also belongs to superclass
- Subclass: more specialized, superclass: more general
- Subclass is derived or inherits from superclass
  - hence, the terms 'derived class' and 'base class'
- In essence:
  1. Objects in the same class have the same set of attributes (different values though) and operations
  2. Objects of subclass have all members of superclass plus some more
  ➔ Objects of a subclass can also be treated as objects of its superclass

```java
Person
  name
  birthday
  getName()

Employee
  name
  birthday
  getName()
  salary
  getSalary()
```
An Employee “is a” Person,
- apart from its own members, salary and getSalary, it also has name, birthday, getName() without having to declare them
- Employee is the subclass (derived) of Person
- Person is the superclass (base) of Employee

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name: String</td>
</tr>
<tr>
<td>- birthday: Date</td>
</tr>
<tr>
<td>+ getName(): String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>- salary: double</td>
</tr>
<tr>
<td>+ getSalary(): double</td>
</tr>
</tbody>
</table>
Inheritance

- Inheritance tree can have many levels
  - A Manager object inherits what an Employee has, including what a Person has.

Only new attributes/operations are listed, inherited attributes/operations are not listed in the subclass's box.
Inheritance in Java

How to?

1. Subclass “extends” superclass
   - New attributes/operations
   - Redefine inherited operations
     - Method overriding

2. Treat subclass objects as superclass objects
   - Access inherited data members and methods
     - Information hiding
   - Initialise inherited data members
     - using constructor of superclass
New attributes/operations

Syntax:

```java
[public] class Subclass extends Superclass {
    /* new features go here */
}
```

Example:

```java
class Employee extends Person {
    private double salary;
    public boolean setSalary(double sal) {
        ...  
        salary = sal;
        return true;
    }
}
```

<table>
<thead>
<tr>
<th>Person</th>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>- name: String</td>
<td>- salary: double</td>
</tr>
<tr>
<td>- birthDate: Date</td>
<td>+ getSalary(): double</td>
</tr>
<tr>
<td></td>
<td>+ getName(): String</td>
</tr>
</tbody>
</table>

Đại học Công nghệ. ĐHQG Hà Nội
New attributes/operations

```java
public class Person {
    private String name;
    private Date birthday;

    public boolean setName(String n) {
        name = n; return true;
    }

    public String getName() {
        return name;
    }
}

public class Employee extends Person {
    private double salary;

    public boolean setSalary(double s) {
        salary = s; return true;
    }

    public double getSalary() {
        return salary;
    }
}
```

//application code
...
Employee e = new Employee();
e.setName("John");
System.out.print(e.getName());
e.setSalary(3.0);

calls to Person's methods from an Employee object
calls to Employee's method from an Employee object
Method overriding

- A subclass can redefine methods inherited from its superclass.
  - To specialise these methods to suit the new problem
- Objects of the subclass will work with the new version of the methods
  - Dynamic binding
- Superclass’s methods of the same name can be reused by using the keyword `super`
Method overriding - Example

Subclass's version does something else

class Animal {
    String name;
    public void sayHello() {
        System.out.println("Huh?");
    }
}
class Cow extends Animal {
    public void sayHello() {
        System.out.println("Mooo...");
    }
}

//client code
Animal a1 = new Animal("Bob");
a1.sayHello();
Cow c1 = new Cow("Alice");
c1.sayHello();
Animal a2 = c1;
a2.sayHello();
More example

Subclass's version calls superclass's version then does something extra

```java
public class Person {
    protected String name;
    protected Date birthday;
    public void display() {
        System.out.print (name + "\," + birthday);
    }
    ...
}

public class Employee extends Person {
    ...
    public void display() {
        super.display();
        System.out.print ("\," + salary);
    }
    }
```
Method overriding - Rules

- New and old version must have the same prototype:
  - Same return type
  - Same argument type
- Private methods cannot be overridden
  - Private members are hidden from subclass
Superclass information hiding

- Superclass programmer and subclass programmer might not be the same person.
- Simple reuse independent of specific implementations
  - Employee does not have to care how name and birthday are stored and processed inside Person but can still use them
- Internal design and implementation of superclass can be modified without requiring changes in subclasses
  - e.g, class Person could have three int attributes instead of one Date attribute for birthday, or two instead of one String for name. Class Employee’s code is not affected.
- *Hiding does not mean preventing source code from being seen by programmers.*
## Access control levels

<table>
<thead>
<tr>
<th>Modifier</th>
<th>accessible within</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>same class</td>
</tr>
<tr>
<td>private</td>
<td>Yes</td>
</tr>
<tr>
<td>package (default)</td>
<td>Yes</td>
</tr>
<tr>
<td>protected</td>
<td>Yes</td>
</tr>
<tr>
<td>public</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**protected access level**

- protected members of a superclass are directly accessible from inside its subclasses.

```java
public class Person {
    protected String name;
    protected Date birthday;
    ...
}

public class Employee extends Person {
    ...
    public String toString() {
        String s;
        s = name + "," + birthday; //no error.
        s += "," + salary;
        return s;
    }
}
```

Subclass can directly access superclass's protected members
In the same package

Default access level is “package”, which means those with “package” access level can be accessed directly from within the same package.

```java
package people;

public class Person {
    String name;
    Date birthday;
    ...
}

package people;

public class Employee extends Person {
    ...
    public String toString() {
        String s;
        s = name + "," + birthday; //no error.
        s += "," + salary;
        return s;
    }
}
```
In different packages

Members with “package” access level **cannot** be accessed directly by subclasses from outside the package.

```java
package people;

public class Person {
  String name;
  Date birthday;
  ...
}

import people.Person;

public class Employee extends Person {
  ...
  public String toString() {
    String s;
    s = name + "," + birthday; //Error.
    s += "," + salary;
    return s;
  }
}
```
In different packages

Members with "protected" access level can be accessed directly by subclasses from outside the package.

```java
package people;

public class Person {
    protected String name;
    protected Date birthday;
    ...
}

import people.Person;

public class Employee extends Person {
    ...
    public String toString() {
        String s;
        s = name + "," + birthday; //no error.
        s += "," + salary;
        return s;
    }
}
```
Inherit from another package

- Can create subclass of a class from a different package
  - Inherit from Java standard library’s classes
  - Inherit classes from third-party vendors

- Inherit without knowing source code
  - Protect source code
  - Increase reusability
Constructor of subclass

- Subclass inherits all attributes/methods of superclass
  - Subclass must initialize inherited members
- But, constructors are NOT inherited
  - syntactically, they have different names
- Two ways to call constructors of the baseclass
  1. ( Implicit) use default constructors
  2. Explicit calls to constructors of the baseclass
class Point {
    protected int x, y;
    public Point() {
    }
    public Point(int xx, int yy) {
        x = xx;
        y = yy;
    }
}

class Circle extends Point {
    protected int radius;
    public Circle() {
    }
}

//client code
Point p = new Point(10, 10);
Circle c1 = new Circle();
Circle c2 = new Circle(10, 10); // error

Default constructor Point() is called

Cannot found Circle(int, int).
Point(int, int) is not inherited
Calling constructors of baseclass

- The initializing baseclass' attributes should be carried out by baseclass' constructors
  - Why?
- Baseclass' constructors can be called using reference `super`
  - Baseclass' constructors must run first
  - If baseclass has no default constructor then its constructor must be called explicitly
class Point {
    protected int x, y;
    public Point() {} 
    public Point(int xx, int yy) {
        x = xx;
        y = yy;
    }
}

class Circle extends Point {
    protected int radius;
    public Circle() {
        super(0, 0);
    }
    public Circle(int xx, int yy, int r) {
        super(xx, yy);
        radius = r;
    }
}

//application code
Point p = new Point(10, 10);
Circle c1 = new Circle();
Circle c2 = new Circle(10, 10, 5); // ok

Explicit calls to
Point(int, int)
class Point {
    protected int x, y;
    public Point(int xx, int yy) {
        x = xx;
        y = yy;
    }
}

class Circle extends Point {
    protected int radius;
    public Circle() {}
    public Circle(int xx, int yy, int r) {
        radius = r;
    }
}

//application code
Point p = new Point(10, 10);
Circle c1 = new Circle();
Circle c2 = new Circle(10, 10, 5);
Constructors - order of execution

```java
class Point {
    protected int x, y;
    public Point() {
        System.out.println("Point constructor");
    }
}

class Circle extends Point {
    protected int radius;
    public Circle() {
        System.out.println("Circle constructor");
    }
}

//application code
Circle c1 = new Circle();
```
Multiple level inheritance

- All classes inherits from class Object
**toString() method**

Inherits from Object class

```java
class Point {
    protected int x, y;
    public String toString() {
        return "<" + x + "," + y + ">";
    }
}

class Circle extends Point {
    protected int radius;
    public String toString() {
        return super.toString() + "," + radius;
    }
}

//application code
Circle c = new Circle();
System.out.println(c);
```

Overriding Object's toString()
New versions are used in System.out.println()
Basic data wrapper types
**final keyword**

- **final attribute**
  - Constant value, assigned value once upon initialisation…a final attribute cannot be changed

- **final method**
  - Cannot be overridden in subclasses

- **final arguments**
  - Method cannot change value of final arguments/parameters

- **final class**
  - Cannot create subclasses of a final class
**final arguments**

class MyDate {
    int year, month, day;
    public MyDate(int y, int m, int d) {
        year = y; month = m; day = d;
    }
    public void copyTo(final MyDate d) {
        d.year = year;
        d.month = month;
        d.day = day;
        d = new MyDate(year, month, day); //error
    }
...
}

Reusing Classes

- Object classes with similar or related attributes and behaviour
  - Person, Student, Manager,…

- Code reuse
  - Copy & paste
    - Manually -> Error-prone
  - Composition – “has-a” relationship
    - the new class is composed of objects of existing classes.
    - reuse the functionality of the existing class, not its form

- Inheritance – “is-a” relationship
  - create a new class as a type of an existing class
  - new class absorbs the existing class's members and extends them with new or modified capabilities
Reusing classes - composition

- Existing class is used as a component of the new class
- Reused features might need new interface
  - Interface must be rewritten when necessary
  - Not flexible enough in some cases

```
Person
- name: String
- birthday: Date

+ Person()
+ getName(): String
```

```
Date
- day: int
- month: int
- year: int

+ Date()
+ nextDate(): Date
+ toString(): String
```

```
Employee
- myself: Person
- salary: double

+ Employee()
+ getName(): String
+ getSalary(): double
...
```

```
Person
- name: String
- birthday: Date

+ Person()
+ getName(): String
```

getName must be rewritten as a wrapper
class Person {
    private String name;
    private Date birthday;
    public String getName() { return name; }
    ...
}

class Employee {
    private Person myself;
    private double salary;
    public String getName() { return myself.getName(); }
    ...
}

class Manager {
    private Employee myself;
    private Employee assistant;
    public String getName() { return myself.getName(); }
    public setAssistant(Employee e) { assistant = e; }
    ...
}

//application code
...
Manager junior = new Manager();
Manager senior = new Manager();
senior.setAssistant(junior);  // error

getName must be rewritten as wrappers

Inflexible! Assistant can’t be a Manager
Reusing classes - inheritance
Inheritance

class Person {
    private String name;
    private Date birthday;
    public String getName() { return name; }
    ...
}

class Employee extends Person {
    private double salary;
    ...
}

class Manager extends Employee {
    private Employee assistant;
    public setAssistant(Employee e) { assistant = e; }
    ...
}

//application code
...
Manager junior = new Manager();
Manager senior = new Manager();
senior.setAssistant(junior); // no error

Yes! Assistant can be a manager