03 - Inheritance and Polymorphism

Modelling (1/2)
- A model should be an abstract yet precise description of a system.
  - Abstract: Hides details
  - Precise: Unambiguous, with a clear meaning
- Models can be graphical or textual descriptions
- The diagrams we have been using are part of the Unified Modelling Language (UML).
  - Class diagrams
  - Object diagrams

Modelling (2/2)
- Model should support reasoning and documentation.
- Choose level of abstraction and scope appropriately

Relation between Classes
- Why think of relationships between classes?
  - Modelling perspective:
    - Manage complexity of the problem and solution
    - Analysis and Documentation.
  - Programming perspective:
    - Code reuse and readability
    - Java provides constructs for supporting some kinds of relationships
- Relationships
  - Dependency
  - Association
  - Aggregation
  - Composition
  - Generalisation/Specialisation

Diagram:
- Car
  - colour: String
  - model: String
  - fuelCapacity: int
  - start(): void
  - stop(): void
  - move(direction: String): void
- Dice
  - colour: String
  - getTop(): int
  - getBottom(): int

Diagram:
- circle1
  - area(): int
  - perimeter(): int

Diagram:
- MyCar: Car
  - circumference(): double
  - area(): double

Diagram:
- Dice ()
  - getTop(): int
  - getBottom(): int

Strength
Dependency

- Defines the “uses” or “knows about” relationship.
- If a class explicitly uses a type introduced by some class, then there is a dependency.
- Indicates that a change in one class may affect the other.
- Dependencies can be given names.
- Weakest relationship between classes

Association

- A specific kind of dependency
- Specifies a structural relation.
- Typically can be conceptually navigated in both directions.
  - Programmatically, this may not be possible
- May have roles
- May have cardinalities

Aggregation and Composition

- Specific kind of association
- Denote “part of” or “has” relationship.
- Composition is stronger than Aggregation
  - Parts are not shared
  - Parts existence depends on existence of the whole

Class Relationships: A Subtle issue

- Sometimes it can be arguable as to whether the relation between classes is a dependency, association, composition, aggregation.
- It is a subtle issue
- For these 4 relations, the impact on the actual Java code is limited.
- If you are not sure, you can use the weaker relationship.
Generalisation / Specialisation

- Denotes the “is-a” or the “is-a-kind-of”
- It is the strongest kind of relation.
- Inheritance is the process of deriving a specialised class from an existing class.
- Java supports inheritance explicitly through several language constructs.

Inheritance: Car Example

- An electric car “is-a” car that runs with an electric engine.
- Car is the superclass or parent-class of ElectricCar
- Car is a generalisation of ElectricCar.
- ElectricCar is a subclass or a derived-class of Car
- ElectricCar is a specialisation of Car.

Inheritance: Student Example

- A PhD student “is-a-kind-of” Student.
  - Unlike other students it has a supervisor, must write a thesis and does not take courses.
  - Like other students, it must register, is in a department, receives a login, aims for a degree.

“Is a” vs “Is a kind of”

- What is the difference?
  - Are there new visible methods fields.
  - Does the visible behaviour change.
- An electric car is a car
  - Has same visible methods and fields
  - Has same visible behaviour (from the user’s perspective)
- A PhD student is a kind of student
  - Has more methods
  - Has different requirements for graduation.
- Practical implications?
  - Substitutability
    - A method using PhDStudents may not work with Students
    - A method using ElectricCar should work with Cars.
  - More on this later...
Inheritance: Lamp Example

- A lamp may be switched on or off by pressing a switch.
- An adjustable lamp inherits all properties of a lamp; in addition it can be dimmed.

```
Lamp
- on: boolean
  +Lamp(boolean initOn)
  +pressSwitch(): void
  +print(): void

AdjLamp
- brightness: double
  +AdjLamp(boolean on)
  +dim(): void
  +print(): void
```

Example - Lamp (2)

```
class Lamp {
  protected boolean on;
  public Lamp(boolean initOn) {
    on = initOn;
  }
  public void pressSwitch() {
    on = !on;
  }
  public void print() {
    System.out.println("Lamp("+on+")");
  }
}

class AdjLamp extends Lamp {
  private double brightness;
  public AdjLamp(boolean on) {
    super(on);
  }
  public void pressSwitch() {
    super.pressSwitch();
    brightness = 1.0;
  }
  public void print() {
    System.out.println("AdjLamp("+on +","+brightness+")");
  }
}
```

Lamp Example - Lessons

- Derived objects...
  - extend their superclass
  - e.g. AdjLamp extends Lamp with dim() and brightness
  - inherit all members of the superclass
  - e.g. All AdjLamp objects have method pressSwitch() and field on
  - can access public and protected fields of the superclass
  - AdjLamp can access on and pressSwitch
  - can override methods of the superclass
  - AdjLamp print() overrides Lamp print()
  - can call public and protected methods of the superclass
  - super(on)
  - super.pressSwitch() //more on this...

- Note difference between inheritance and visibility!
  - If CheapAdjLamp is a subclass of AdjLamp, its objects cannot access brightness even though they do have a brightness field.

Super

- Used to call the constructor of a superclass
  - can only be called from subclass constructor.
  - must be called on the first line.
  - can be called with different parameters according to the superclass constructor you wish to call.
    - super(on):
    - super():
    - super(arg1, arg2):

- Used to call an overridden method from a superclass.
  - super.pressSwitch()
Inheritance: Watch Example

public class Watch {
    protected double _price;
    protected String _model;

    public Watch(String model, double price) {
        _model = model;
        _price = price;
    }

    public double setSalePrice() {
        _price = _price * 0.80;
        return _price;
    }
}

public class LuxuryWatch extends Watch {
    protected double _premium;

    public LuxuryWatch(String model, double price, double premium) {
        super(model, price);
        _premium = premium;
    }

    public double setSalePrice() {
        return super.setSalePrice() + _premium;
    }
}

Visibility Revisited: Protected

- Remember Public and Private keywords?
- A new one: Protected
  - Protected methods and fields can be accessed from the class and subclasses.
  - Less restrictive than private (only allows access from within class)
  - More restrictive than public (allows everyone to access)
- E.g.
  - Subclass of AdjLamp cannot access fields brightness, but can on.
  - Other classes cannot access either.

Visibility: Summary

<table>
<thead>
<tr>
<th></th>
<th>public</th>
<th>protected</th>
<th>package (default)</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>within class</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>within package</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>within subclass</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Anywhere</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Using Derived Classes

- Derived classes are used exactly in the same way as normal classes.
  
  ```java
  AdjLamp l = new AdjLamp(true);
  l.print();
  l.dim();
  l.pressSwitch();
  ...
  ```

- In addition, if an adjustable lamp is a kind of lamp, we can also do:
  
  ```java
  Lamp l = new AdjLamp(true);
  ```

- However, there are some details we need to discuss...

Using Derived Classes: Polymorphism

```java
public class Room {
  private Lamp l;
  public Room(Lamp l) {this.l = l;};

  public void strobe(int pulses) {
    for (int i=0; i<= pulses; i++) {
      l.print();
      l.pressSwitch();
      l.print();
      l.pressSwitch();
    }
  }

  public void nightclub() {
    Room r = new Room(new AdjLamp(false));
    r.strobe(40);
  }
}
```

Which `print()` method gets called?

Late Binding

- Late binding means that the bind between method call and the actual method to be executed is at runtime.

```java
public class Room {
  private Lamp l;
  public Room(Lamp l) {this.l = l;};

  public void strobe(int pulses) {
    for (int i=0; i<= pulses; i++) {
      l.print();
      l.pressSwitch();
      l.print();
      l.pressSwitch();
    }
  }

  public void nightclub() {
    Room r = new Room(new AdjLamp(false));
    r.strobe(40);
  }
}
```

At runtime, the JVM will access the object (which has actual type `AdjLamp`), and call `print()` and `pressSwitch()` accordingly.

**Apparent and Actual types**

- The *apparent type* of an object is that which the compiler can statically infer.
  
  - e.g. `Lamp l = ...`

- The *actual type* is the type the object really has.
  
  - e.g. `new AdjLamp(...)`

**Rules**

- Apparent type must be super type of the actual type
  
  - e.g. `AdjLamp l = new Lamp() //WRONG. Compile Error.`

- Access methods and fields according to apparent type
  
  - e.g. `Lamp l = ... l.dim() //WRONG. Compile Error`

- Java checks these rules at compile time.
  
  - It is a strongly typed language
  
  - It does static type checking.
Binding and Typing issues: Example

```java
public static void main(String[] args) {
    Lamp l = new Lamp(true);
    l.print();
    l.pressSwitch();
    l.print();
    // l.dim();     // ERROR
    System.out.println("Adjustable");
    AdjLamp adjL = new AdjLamp(false);
    adjL.print();
    adjL.pressSwitch();
    adjL.dim();
    adjL.print();
    System.out.println("Assignment");
    l = new AdjLamp(false);
    l.print();
    l.pressSwitch();
    l.print();
    // l.dim();  // ERROR
}
```

OUTPUT

```
Lamp( true)
Lamp( false)
Adjustable
AdjLamp( false,  1.0 )
AdjLamp( true,  0.9 )
Assignment
AdjLamp( false,  1.0 )
AdjLamp( true,  1.0 )
```

Substitution Principle

- In general, inheritance can be used when the (impure) substitution principle holds:
  - I can replace the generalised class with a more specialised class, and the program will not break.
  - Subclasses extend/change the behaviour of the superclass.
  - Provides flexibility but can also introduce runtime errors.
  - E.g. An amphibious car is an impure substitution of a car (because it extends them, with aquatic capabilities)
- Some people would argue that inheritance should only be used when the pure substitution principle holds:
  - I can replace the specialised class with a more general class, and the program will not break.
  - More restrictive.
  - Does not allow subclasses to extend their superclass, only change the internal implementation.
  - E.g. A car is a pure substitution of a PetrolCar

Upcasting

```
public class Room {
    private Lamp l;
    public Room(Lamp l) {this.l = l;}
    public void strobe(int pulses) {
        for (int i=0; i<= pulses; i++) {
            l.print();
            l.pressSwitch();
            l.print();
        }
    }
}
```

The process of treating the object of type AdjLamp as if it were of type Lamp is called upcasting. Upcasting is done automatically.

Downcasting

```
public class Room {
    private Lamp l;
    public Room(Lamp l) {this.l = l;}
    public void dimStrobe(int pulses) {
        for (int i=0; i<= pulses; i++) {
            ((AdjLamp) l).dim(); // Downcasting
            l.pressSwitch();
        }
    }
}
```
Downcasting: Example

```java
public static void main(String[] args) {
    Lamp l = new AdjLamp(true);
    l.print();
    l.pressSwitch();
    // l.dim(); // ERROR
    System.out.println("About to downcast");
    AdjLamp adjL = (AdjLamp) l; // downcast
    adjL.print();
    adjL.pressSwitch();
    adjL.dim();
    adjL.print();
    l = new Lamp(false);
    System.out.println("About to downcast");
    adjL = (AdjLamp) l; // downcast
    l.pressSwitch();
    l.print();
}
```

OUTPUT

```
AdjLamp(true, 1.0)
AdjLamp(false, 1.0)
About to downcast
AdjLamp(false, 0.9)
About to downcast
ClassCastException
```

Downcasting: Lessons

- **Downcasting** or **narrowcasting** is an explicit type conversion
- Some downcasts cannot be verified at compile time, and may produce run-time errors (ClassCastException).
- If we write \((t)v\) where \(t\) is a type and \(v\) a variable
  - If the actual type of the object referenced by \(v\) is \(t\) or a subclass of \(t\) then \((t)v\) returns a reference of type \(t\) to the object referenced by \(v\)
  - If not, Java raises a ClassCastException, which means that the object pointed by \(v\) cannot be cast onto \(t\)

Downcasting: Bag Example (1/2)

```java
class Thing {
    public void print() {
        System.out.println("this is a something");
    }
}
class Coin extends Thing {
    protected int _value;
    public Coin(int val) {_value = val; }
    public int getValue() { return _value; }
    public void print() { System.out.println("value = " + _value + " coin"); }
}
```

Downcasting: Bag Example (2/2)

```java
class Bag {
    private Thing[] _things = new Thing[10];
    private int _index = 0;
    public void add(Thing t) {_things[_index++] = t; }
    public Thing remove1() { _index--; Thing t = _things[_index]; _things[_index] = null; return t; }
}
```

```java
public static void main(String[] args) {
    Bag b = new Bag();
    b.add(new Coin(5));
    Thing t = b.remove1();
    t.print(); // compile error (obviously)
    Coin c = (Coin) t;
    c.getValue(); // ok
}
```
### Polymorphism Summary

**Polymorphism** refers to the ability to treat uniformly structures which may have more than one form.

**Related concepts**
- Inheritance
- Actual and Apparent types
- Casting (up and down)
- Late binding
- Pure and impure substitution
- Static type checking and run-time casting errors

---

### Printing Queue - Document Superclass

```java
public class Document {
    protected String _name;
    protected String _content;
    public Document (String n, String c) {_name = n; _content = c;}
    public void print() {System.out.println(_content); }
    public String getName() {return _name; }
    public boolean equals(Document d) { //Note that content is ignored
        return(d.getName().equals(_name)); }
}
```

---

### Printing Queue Class Diagram

```
public class Document {
    protected String _name;
    protected String _content;
    public Document (String n, String c) {_name = n; _content = c;}
    public void print() {System.out.println(_content); }
    public String getName() {return _name; }
    public boolean equals(Document d) { //Note that content is ignored
        return(d.getName().equals(_name)); }
}
```
Printing Queue - Document Subclasses

```java
public class MSWordDocument extends Document {
    MSWordDocument (String n, String c) {
        super(n, c);
        public void print() {
            System.out.println("MSWord-" + _name);
        }
    }
}
```

```java
public class PDFDocument extends Document {
    PDFDocument(String n, String c) {
        super(n, c);
        public void print() {
            System.out.println("Acroread-" + _name);
        }
    }
}
```

```java
public class PSDocument extends Document {
    PSDocument(String n, String c) {
        super(n, c);
        public void print() {
            System.out.println("PSDocument-" + _name);
        }
    }
}
```

Printing Queue - Main()

```java
public class PrintQueue {
    private LinkedList _docList = new LinkedList();
    public void add(Document d) { _docList.add(d); }
    public void print() {
        while (!_docList.isEmpty()) {
            Document d = (Document) _docList.removeFirst();
            d.print();
        }
    }
    public static void main(String[] args) {
        PrintQueue p = new PrintQueue();
        p.add(new MSWordDocument("Hello World.doc", "Hi!"));
        p.add(new PDFDocument("Lecture1.pdf", "Introduction"));
        p.add(new PSDocument("Lecture2.ps", "Inheritance"));
        p.print();
    }
}
```

Inheritance hierarchies

```
java.lang.Object

... String Random CollegeMember

... Student StaffMember

... Undergrad PhdStudent MscStudent Staff Member

Lecturer Professor

Student Staff

Note: Lecture inherits indirectly from Staff.

College Member

Note that the hierarchy need not be a comprehensive classification:
- e.g. Readers are academics that are not lecturers nor professors
- e.g. Some researchers are not academics nor temporary staff
```

The Object Class

- **Object** is THE superclass.
- In Java everything is a subclass of Object.
- Being an Object subclass is not declared explicitly!
The Java Class Hierarchy

Object
  ↓
Class
  ↓
String
  ↓
Number
  ↓
Byte
  ↓
Float
  ↓
Long
  ↓
Short

also see Java Development Kit Documentation

The Java Type Hierarchy

Multiple Inheritance

- In Java a class can only have one direct superclass
- Some object oriented languages support multiple inheritance: A class can have more than one direct superclass.

PartTimePhDandResearcher
  ↓
PhDStudent
  ↓
Researcher

Pure or impure substitution?

- Java provides "interfaces" and interface hierarchies to make up for it. //see later...

Some comments on inheritance

- Useful when used correctly!
- Do not get carried away!
  - Think of the pure substitutability principle as a guideline...
  - In practice you sometimes may need break guidelines...
    - using impure substitutability,
    - or even non-substitutability (very exceptionally!)
- How to decide when to bend design rules?
  - Practice and Experience...
  - Common sense...
  - Reading good programmer's code...
  - Reading on design...
- Some examples to think about
  - Should PieSlice extend Circle?
  - Should CarParkingLot extend VehicleParkingLot?
  - Should Square extend Rectangle?
A Square IS A Rectangle?

Rectangle r = new Square();
r.setHeight(2);
r.setWidth(3);
System.out.println(r.perimeter());

- The answer is No!
  - A square is like a rectangle.
  - This is impure substitution

The “IS A” relation in OO is refers to visible behaviour.

Debate on pure substitutability

- In principle, A should be a subclass of B only if
  - the pure substitutability principle holds...
  - A “is a” B

- Does this hold for Square and Rectangle?
  - i.e. a Square “is a” Rectangle?
  - i.e. Is the behaviour of square like that of a rectangle?
  - i.e. If I think I am operating on an object I believe is of class Rectangle, but it turns out to be of class square, would I notice the difference?

Using inheritance...

- Useful when used correctly!
- Do not get carried away!
- Real systems rarely have deep inheritance hierarchies
- Try to use pure substitutability as a guideline...
- However, in practice you these guidelines are broken...
- How to decide when to bend design rules?
  - Practice and Experience...
  - Common sense...
  - Reading good programmer’s code...
  - Reading on design...
Checklist

- UML and modelling
- Class relationships: Dependency, Association, Aggregation, Composition and Inheritance
- Keywords super and extends
- Visibility: public, private, protected and default (package)
- "Is-a" vs. "Is-a-kind-of"
- Polymorphism
- Actual and Apparent types
- Late binding, Static type checking and ClassCastExceptions
- Casting (up and down)
- instanceof
- Pure and impure substitution
- Class hierarchies and the Object class.