Inheritance & Polymorphism

Object-oriented programming
Outline

- Polymorphism
  - upcasting/downcasting
  - Dynamic linking
- Abstract classes and methods
- Multi-inheritance & interfaces
- Design patterns
  - Prototype, Template method

- Readings:
  - *Java how to program*, chapter 10
What is polymorphism?

- Polymorphism: *exist in many forms*

- Polymorphism in programming
  - Function polymorphism: same name, different arguments
  - Object polymorphism:
    - An object can be treated in different ways
      - A Manager object can be seen as an Employee object as well
    - Different objects interpret the same message differently
      - How do kangaroos and frogs "jump"?
**Upcasting**

- **Upcasting**: cast "up" the inheritance diagram
- Take an object reference and treat it as if it refers to its base type

  ```java
  Cat c = new Cat("Tom");
  Animal a = c; // upcasting
  
  // Cannot invoke methods not provided by the base type
  a.chaseTail(); // Error! method not found in class Animal
  
  // But
  a.makeASound(); // "Meow...", Cat's makeASound() gets to run
Downcasting

- Cast "down" the inheritance diagram

```java
Animal animal = new Cat("Tom"); // upcast ...
Cat c = (Cat) animal; // downcast
c.sayHello(); // "Meow.."
```

→ But, not always possible. Why?

```java
Animal animal = new Animal("Dummy");
...
Cat c = (Cat) animal; // no compile-time error
c.chaseTail(); // run-time error
```
Polymorphism

The same message is interpreted differently depending on the object's type

We can send a message to an object and let the object figure out the right thing to do, without us taking care of which derived class the object belongs to.
class Animal {
    ...
    public void makeASound() {
        System.out.print("Uh oh!");
    }
}
class Cat extends Animal {
    ...
    public void makeASound() {
        System.out.print("Meow...");
    }
}
class Cow extends Animal {
    ...
    public void makeASound() {
        System.out.print("Moo...");
    }
}

Animal myPets[] = new Animal[2];
myPets[0] = new Cat("tom");
myPets[1] = new Cow("mini");
for (int i = 0; i < myPets.length; i++) {
    myPets[i].makeASound();
}
class Animal {
    ...
    public void makeASound() {
        System.out.print("Uh oh!");
    }
    public void introduce() {
        makeASound();
        System.out.println(" I'm " + name);
    }
}

class Cat extends Animal {
    ...
    public void makeASound() {
        System.out.print("Meow...");
    }
}

class Cow extends Animal {
    ...
    public void makeASound() {
        System.out.print("Moo...");
    }
}

Animal pet1 = new Cat("Tom Cat");
Animal pet2 = new Cow("Mini Cow");
pet1.introduce();
pet2.introduce();

Meow... I'm Tom Cat
Moo... I'm Mini Cow
class Animal {
    ... public void makeASound() {
        System.out.print("Uh oh!");
    }
    public void introduce() {
        makeASound();
        System.out.println("I'm " + name);
    }
}

class Dog extends Animal {
    public void makeASound() {
        System.out.print("Bow wow...");
    }
}

class Duck extends Animal {
    public void makeASound() {
        System.out.print("Quack quack...");
    }
}

You can add as many new animal types as you want without modifying the introduce( ) method!

Separate things that change from things that stay the same.
Dynamic & static binding

- Method binding: connect a method call to a method body
- Static/early binding: performed by compiler/linker before the program is run.
  - The only option of procedural languages.
- Dynamic/late binding: performed during run-time
  - Java uses late binding, except for static, final, and private methods.
    - private methods are implicitly final.
class Animal {
    ... 
    private void makeASound() {
        System.out.print("Uh oh!");
    }
    public void introduce() {
        makeASound();
        System.out.println(" I'm " + name);
    }
}

class Cat extends Animal {
    ... 
    private void makeASound() {
        System.out.print("Meow...");
    }
}

class Cow extends Animal {
    ... 
    private void makeASound() {
        System.out.print("Moo...");
    }
}

Animal pet1 = new Cat("Tom the Cat");
Animal pet2 = new Cow("Mini the Cow");
pet1.introduce();
pet2.introduce();

Uh oh! I'm Tom the Cat
Uh oh! I'm Mini the Cow
Abstract class

- Sometimes we don't want objects of a base class to be created

Examples:
- Animal, Cat, Cow, Dog,…
  - An Animal object makes no sense
  - What sort of sound does it make?

- Shape, Point, Rectangle, Triangle, Circle
  - What does a generic Shape look like?
  - How to draw it?

Solution: make it an abstract base class
abstract class Shape {
    protected int x, y;
    Shape(int _x, int _y) {
        x = _x;
        y = _y;
    }
}

class Circle extends Shape {
    private int r;
    public Circle(int _x, int _y, int _r) {
        super(_x, _y);
        r = _r;
    }
    ...
}

Shape s1 = new Circle();
Shape s = new Shape(10, 10) // compile error

Abstract class: objects cannot be instantiated
Concrete class: objects can be instantiated
Abstract method

- Sometimes we want a method in base class to serve as the common interface of subclasses' versions only
  - i.e. it should never be called
  - e.g. Animal.makeASound(). It contains only dummy code
- Solution: make it an abstract method
  - An abstract method has only a declaration and no method body (i.e. no definition).
    - abstract void f();
  - The class containing an abstract method must be qualified as abstract
  - An abstract method must be overriden and defined in a derived class so that objects of that class can be created (concrete class)
abstract class Shape {
    protected int x, y;
    Shape(int _x, int _y) {
        x = _x;
        y = _y;
    }
    abstract public void draw();
    abstract public void erase();
    public void moveTo(int _x, int _y) {
        erase();
        x = _x;
        y = _y;
        draw();
    }
}

class Circle extends Shape {
    private int r;
    public Circle(int _x, int _y, int _r) {
        super(_x, _y);
        r = _r;
        draw();
    }
    public void draw() {
        System.out.println("Draw circle at ("+x+","+y")");
    }
    public void erase() {
        System.out.println("Erase circle at ("+x+","+y")");
    }
}
Design pattern: Template method

abstract class Shape {
  protected int x, y;

  public void moveTo(int x1, int y1) {
    erase();
    x = x1;
    y = y1;
    draw();
  }

  abstract public void erase();
  abstract public void draw();
}
Abstract super class

- As a super class
  - A common superclass for several subclasses
  - Factor up common behavior
  - Define the methods all the subclasses respond to

- As an abstract class
  - Force concrete subclasses to override methods that are declared as abstract in the super class
    - Circle, Triangle must implement their own draw() and erase()
  - Forbid creation of instants of the abstract superclass
    - Shape objects are not allowed
Clever Factoring Style

- **Common Superclass**
  - Factor common behavior up to the superclass
  - Superclass sends itself messages to invoke various parts of the behavior
    - Will rely on the “pop-down” behavior to work correctly!

- **Special subclasses**
  - As short as possible
  - Rely on the superclass for common behavior
  - Override key methods to customize behavior with minimal code
    - May use super.foo()
  - Rely on pop-down behavior to do the right thing!
Account example

- Problem details:
  - You need to store information for bank accounts
  - Assume that you only need to store the current balance, and the total number of transactions for each account.
  - The goal for the problem is to avoid duplicating code between the three types of account.
  - An account needs to respond to the following messages:
    - constructor(initialBalance)
    - deposit(amount)
    - withdraw(amount)
    - endMonth()
  - Apply the end-of-month charge, print out a summary, zero the transaction count.
Account example

Types of Accounts

- Normal
  - Fixed $5.0 fee at the end of the month

- Nickle ‘n Dime
  - $0.50 fee for each withdrawal charged at the end of the month

- Gambler
  - With probability 0.49 there is no fee
  - With probability 0.51 the fee is twice the amount withdrawn
Design process

- Factoring
  - Put common behavior in one place
  - Subclasses are used to implement the specific deviation from the common behavior

- Abstract methods
  - Provide prototypes for concrete methods to be implemented by subclasses
Class design diagram

Account
*balance
*transactions
-deposit
-withdraw
-endMonth
-endMonthCharge (abstract)

Fee
-endMonthCharge

NickleNDime
*withdrawCount
-withdraw
-endMonthCharge

Gambler
-withdraw
-endMonthCharge
Multi-inheritance

**Animal**
- name
+ makeASound()

**Cat**
+ makeASound()

**Cow**
+ makeASound()

**CanFly**
- topspeed
+ fly()

**CanFight**
- strength
+ fight()

**SuperCow**
Java interfaces

- Java does not support multiple inheritance
  - This is often problematic
    - What if we want an object to be multiple things?

- Interfaces
  - A special type of class which
    - Defines a set of method prototypes
    - Does not provide the implementation for the prototypes
    - Can also define final constants
Java interfaces

- A Class
  - Can “extend” only one class, i.e. only ONE superclass
  - Can “implement” MULTIPLE interfaces!

- Class Server implements Pingable
  - Server is a class
  - It implements the Pingable interface
  - Server MUST provide implementations for all the method prototypes in the Pingable interface
  - A Server object can serve as a substitute wherever we want a Pingable Object.
    - Similar to a superclass
Java interfaces

- Lightweight
  - Allow multiple classes to respond to a common set of messages but without the implementation complexity.

- Similar to Subclassing but...
  - Good news
    - Class has only one superclass
    - Can implement multiple interfaces
  - Bad news:
    - Interface only gives the method prototype and not the implementation
Interface example

- Special keyword ‘interface’
- Similar to defining a class, but instead use the keyword interface
- Methods are empty (no { and } or code)
- Example

```java
public interface Moodable {
    public Color getMood();
    // interface defines getMood() prototype
    // but no code
}
```
Implementing an Interface

- “implements” keyword
  - Similar to extend, but followed by a comma separated list
- Example

```java
public class Student implements Moodable {
    public Color getMood() {
        if (getStress()>100) return(Color.red);
        else return(Color.green);
    }
    // rest of Student class stuff as before...
```
interface Action {
    void moveTo(int x, int y);
    void erase();
    void draw();
}

class Circle1 implements Action {
    int x, y, r;
    Circle1(int _x, int _y, int _r) {
    ...}
    public void erase() {...}
    public void draw() {...}
    public void moveTo(int x1, int y1) {...}
}

class ImageBuffer {
...
}

class Animation extends ImageBuffer implements Action {
...
    public void erase() {...}
    public void draw() {...}
    public void moveTo() {...}
}
```java
interface CanFight {
    void fight();
}
interface CanSwim {
    void swim();
}
interface CanFly {
    void fly();
}
class ActionCharacter {
    public void fight() {...}
}
class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
    public void swim() {...}
    public void fly() {...}
}
```
public class Adventure {
    public static void t(CanFight x) { x.fight(); }
    public static void u(CanSwim x) { x.swim(); }
    public static void v(CanFly x) { x.fly(); }
    public static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Treat it as a CanFight
        u(h); // Treat it as a CanSwim
        v(h); // Treat it as a CanFly
        w(h); // Treat it as an ActionCharacter
    }
}
Extending an interface with inheritance

```java
interface Monster {
    void menace();
}
interface Lethal {
    void kill();
}
interface Vampire extends Monster, Lethal {
    void drinkBlood();
}

class VeryBadVampire implements Vampire {
    public void menace() {...}
    public void kill() {...}
    public void drinkBlood() {...}
}
```
Conflict (1)

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C {
    public int f() { return 1; }
}
class C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; } // overloaded
}
class C3 extends C implements I2 {
    public int f(int i) { return 1; } // overloaded
}
Conflict (2)

```java
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C {
    public int f() { return 1; }
}

class C4 extends C implements I3 {
    // Identical, no problem:
    public int f() { return 1; }
}

class C5 extends C implements I1 {...} //error

interface I4 extends I1, I3 {...} //error
```
Cloning objects

```java
class Animal {
    String name;
    public Animal( String name_) { name = name_; }
    public Animal(Animal b) { name = b.name; }
    public void sayHi() { System.out.println( "Uh oh!"); }
}
class Cat extends Animal {
    public Cat(String name_) { super(name_); }
    public Cat(Cat d) { super(d); }
    public void sayHi() { System.out.println( "Meow..." ); }
}

Cat cat = new Cat("Tom");
Cat c = new Cat(cat); c.sayHi();
Animal a = new Animal(cat); a.sayHi();
```

Meow...
Uh oh!
Cloning objects

Cat cat = new Cat("Tom");
Cat c = new Cat(cat); c.sayHi();
Animal a = new Animal(cat); a.sayHi();

- How to clone objects without knowing their actual type?
  - copy constructor? Nope!
  - copy method?
    - interface Cloneable and clone()
Method clone()

class Animal {
    String name;
    public Animal( String name_) { name = name_; }
    public Animal(Animal b) { name = b.name; }
    public Animal clone() { return new Animal(this); }
    public void sayHi() { System.out.println( "Uh oh!"); }
}
class Cat extends Animal {
    public Cat(String name_) { super(name_); }
    public Cat(Cat d) { super(d); }
    public Cat clone() { return new Cat(this); }
    public void sayHi() { System.out.println( "Meow..."); }
}

Cat cat = new Cat("Tom");
Cat c = cat.clone(); c.sayHi();
Animal a = cat.clone(); a.sayHi();
Design pattern: Prototype

Client
Operation()

Prototype
Clone()

ConcretePrototype1
Clone()
return copy of self

ConcretePrototype2
Clone()
return copy of self

\( p = \text{prototype} \rightarrow \text{Clone()} \)