Abstract classes

Object-Oriented Programming
Outline

- Abstract classes
- Abstract methods
- Design pattern: Template method
- Dynamic & static binding
- Upcasting & Downcasting

Readings:
- HFJ: Ch. 8.
- GT: Ch. 8.
Our previous design

Dog  d = new Dog();
Cat  c = new Cat();

Fine. But...

Animal  anim = new Animal();

What does an Animal look like?
What does an Animal look like?

- What does a new Animal() object look like?
- What are the values of its instance variables?
- What should makeNoise(), eat(), and roam() do?

- Do we ever need an Animal object?
What does a Shape look like?

- What does a generic Shape object look like?
- How to `draw()` it?
- Do we ever need a Shape object?
Abstract classes

- Some classes just should **not** be instantiated!
  - We want Circle and Triangle objects, but no Shape objects.
  - We want Dogs and Cats, but no Animal objects…

- Make those generic classes **abstract** classes
  ```java
  abstract class Animal { ... }
  ```

- The compiler will guarantee that no instances of abstract classes are created.
- But object references of abstract class types are allowed.

```java
Animal a = new Animal(); // Error!!!
Animal anim = new Dog(); // no error.
```
abstract public class Animal {
    public void eat() {}
    ...
}

public class MakeAnimal {
    public void go() {
        Animal a;
        a = new Hippo();
        a = new Animal();
        a.eat();
    }
}

This is OK. You can always assign a subclass object to a super class reference, even if the superclass is abstract.

class Animal is marked abstract, so the compiler will NOT let you do create an instance of Animal.

% javac MakeAnimal.java
MakeAnimal.java:5: Animal is abstract; cannot be instantiated
    a = new Animal();
    ^
1 error
A class that is not abstract is called a **concrete** class.

How do we know when a class should be abstract?
Abstract vs. Concrete

- mobile phone
- smart phone
- iPhone
- iPhone 4
- iPhone 4S
Abstract methods

- How do we implement?
  - Animal.makeNoise(), eat()...
  - We can't think of a generic implementation that is useful

- So, we mark those methods **abstract**.
- Abstract methods has no body.

```java
abstract public class Animal {
    public abstract void makeNoise();
    ...
}
```

`public void makeNoise() {
    System.out.print("Hmm");
}`

No method body! End it with a semicolon.
Abstract methods

- If you declared a method abstract, you must mark the class abstract, as well. You can't have a concrete class with an abstract method.

- An abstract class means that it must be extended.
- An abstract method means that it must be overridden.

- A concrete subclass must have all the inherited abstract methods implemented.
abstract public 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;
    }
}

public class Circle extends Shape {
    private int radius;
    public Circle(int _x, int _y, int _r) {
        super(_x, _y);
        radius = _r;
    }
    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();
}
Account example

- You need to store information for bank accounts: 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.
- The end-of-month charge is calculated depending on types of Accounts
  - *Normal*: Fixed $5.0 fee at the end of the month
  - *Nickle ‘n Dime*: $1.00 fee for each withdrawal charged at the end of the month
  - *Gambler*:
    - With probability 0.49 there is no fee and no deduction to the balance
    - With probability 0.51 the fee is twice the amount withdrawn
Class design diagram

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

Fee
-endMonthCharge

NickleNDime
*withdrawCount
-withdraw
-endMonthCharge

Gambler
-withdraw
-endMonthCharge
public class AnimalList {
    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public void add(Animal a) {
        if (nextIndex < animals.length) {
            animals[nextIndex] = a;
            System.out.println("Animal added at "+nextIndex);
            nextIndex++;
        }
    }
}

public class AnimalTestDrive {
    public static void main(String[] args) {
        AnimalList list = new AnimalList();
        Dog d = new Dog();
        Cat c = new Cat();
        list.add(d);
        list.add(c);
    }
}
public class AnimalList {
    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public Animal get(int index) {
        return animals[index];
    }

    public void add(Animal a) {
        if (nextIndex < animals.length) {
            animals[nextIndex] = a;
            System.out.print("Animal added at "+ nextIndex);
            nextIndex++;
        }
    }
}

public class DogTestDrive {
    public static void main(String[] args) {
        AnimalList list = new AnimalList();
        Dog d = new Dog();
        list.add(d);
        d = list.get(0); // Error!
        d.chaseCats();
    }
}

% javac DogTestDrive.java
DogTestDrive.java:6: incompatible types found : Animal
required: Dog
    d = list.get(0);
   ^

The compiler doesn't know that list.get() refers to a Dog object!
public class AnimalList {
    private Animal[] animals = new Animal[5];
    private int nextIndex = 0;

    public Animal get(int index) {
        return animals[index];
    }

    public void add(Animal a) {
        if (nextIndex < animals.length) {
            animals[nextIndex] = a;
            System.out.print("Animal added at "+nextIndex);
            nextIndex++;
        }
    }

    public static void main(String[] args) {
        AnimalList list = new AnimalList();
        Dog d = new Dog();
        list.add(d);
        Animal a = list.get(0); // We know the object is a Dog!
        a.chaseCats(); // Error! Animal doesn't have chaseCats()!
    }
}

The compiler doesn't know that a refers to a Dog object!
Subclass object & the inherited part

Cow c = new Cow();
Object o = c;

a single object on the heap:
The Cow object.
It contains both the Cow part and the Object part

Cow
Object
equals()
getClassName()
hashCode()
toString()
moo()
implemented features
Cow's specific feature

The Object remote control has fewer buttons!
The rules

- Which method version get invoked depends on the object type.
- Whether a method call is allowed depends on the reference type – *what buttons the remote control has*.

```
Object o = new Cow();
o.toString();
o.moo();
```

- Cow's `toString()` is invoked because `o` is now referring to an `Cow` object.
- `o.toString()` is allowed because `Object` has `toString()`. But `o.moo()` is not allowed because `Object` does not has `moo()`
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.
Casting

- How to make the Cow act like a Cow, again?
  - Use an `explicit` cast:

```java
Object o = new Cow();
o.toString();
o.moo(); // Error!
```

- `():` cast operator, casting to the type `Cow`

- Explicit cast is not always possible:

```java
Object o = new Cat();
Cow c = (Cow) o; // no compile-time error
c.moo(); // run-time error
```
Upcasting & down casting

- **Upcasting**: casting *up* the diagram.
- **Downcasting**: casting *down* the diagram.

- Object o = new Cow();

  *implicit casting from Cow to Object*

  ```java
  implicit casting from Cow to Object
  ```

- Object o;
  
  ```java
  Cow c = (Cow) o;
  ```

  *explicit casting from Object to Cow*
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