Intro to Computer Science & Programming

A Cranes Club Initiative
Object-Oriented Programming and Design

Data Structures
Object-Oriented Programming and Design

- Inheritance
- Abstract classes
- Polymorphism
- Interfaces
- Encapsulation
- Object composition
Inheritance

- The ability for a **parent class** (base class) to pass down behaviors and properties to a **child class** (sub class).
- Child class may introduce new methods or **override** old ones.
- The **extends** keyword. A child class **extends** a parent class.
- All classes implicitly extend the **Object class**
Some Keywords

- **public** - all classes can access it.
- **protected** - all descendants and classes in the same package can access it.
- **private** - no other classes can access it.
- **no keyword** - classes within the same **package** can access it.
- **static** - only one, not many. Class method/variable vs. instance method/variable.
Inheritance

class Mammal {
    private static int count = 0;
    protected int height, weight;
    protected boolean backbone, heart, lungs, stomach, eyes, mouth, nose, ears;

    protected String name;

    Mammal()
    {
        count++;
    }

    protected void eat(){}
    protected void sleep() {}
    protected void procreate() {}
    protected void die()
    {
        count--;
    }
}

class Dog extends Mammal {
    protected boolean muzzle, loyalty;

    protected void bark(){}
    protected void chaseTail() {}
    protected void wagTail() {}
    protected void dig() {}
}

class Cat extends Mammal {
    protected boolean claw, ego;

    protected void meow(){}
    protected void catchRodents() {}
    protected void lickFur() {}
    protected void spitHairBall() {}
}
Human Beings

class Homonidae extends Primates {
    protected static final boolean legs LongerThanArms = true,
    opposeableThumbs = true, noTail = true;
    void walkOnTwos() {}
    void makeTools() {}
}

final class HumanBeings extends Homonidae {
    // However, man is created to attain his perfection not only
    // through the dominion and autonomy of the Principle itself,
    // but also by accomplishing his own portion of responsibility
    // in passing through this period
    public static final boolean portionOfResponsibility = true;
}
Multiple Inheritance

- The ability to inherit from multiple classes.
- In Java (and several other OOP languages), this is **not allowed**.
Let’s Illustrate Using DP

The Principle of Restoration through Indemnity

Restoration Through Indemnity
Position of fallen people

Before the fall
- God
- One in heart
- Humans
  - Relate only with God

After the fall
- God
- Satan
- Creator
- Kinship of blood
- Humans
  - Midway position
Let’s Illustrate Using DP

- Which version of `offering()` will this call?

```java
class God {
    void offering()
    {
        System.out.println("I live for the sake of others.");
    }
}

class Satan {
    void offering()
    {
        System.out.println("I live for the sake of myself!");
    }
}

class FallenHuman extends God, Satan{
}
FallenHuman adam = new FallenHuman();
adam.offering();
```
Abstract Classes

- A class that’s meant to be an outline, not an implementation.
- Cannot create objects of abstract class.
- May be partially implemented.
- Must use the `abstract` keyword to declare the class and any abstract methods.
Let’s Revisit Mammal

```java
class Mammal {
    private static int count;
    protected int height, weight;
    protected boolean backbone, heart,
    lungs, stomach, eyes, mouth, nose, ears;

    protected String name;

    Mammal() {
        count++;
    }

    protected void eat(){}
    protected void sleep() {}
    protected void procreate() {}
    protected void die()
    {
        count--;
    }
}

Mammal a;
a = new Mammal();
```

```java
abstract class Mammal {
    private static int count;
    protected int height, weight;
    protected boolean backbone, heart,
    lungs, stomach, eyes, mouth, nose, ears;

    protected String name;

    Mammal() {
        count++;
    }

    abstract protected void eat();
    abstract protected void sleep();
    abstract protected void procreate();
    protected void die()
    {
        count--;
    }
}

Mammal a; //allowed
a = new Mammal(); //not allowed!
```
Polymorphism

- The ability to dynamically assign an object’s behavior.
- Child classes override methods of base classes.
- Allows for code reuse, better organization and easier maintenance.
abstract class Pet {
  String _name = "";
  Pet(String name) {
    _name = name;
  }
  abstract void makeSound();
}

class Dog extends Pet {
  Dog() {
    super("Fido");
  }
  Dog(String name) {
    super(name);
  }
  void makeSound() {
    System.out.println("Woof woof!");
  }
}

class Cat extends Pet {
  Cat() {
    super("Griz");
  }
  Cat(String name) {
    super(name);
  }
  void makeSound() {
    System.out.println("Meow meow!");
  }
}

Pet a;
a = new Dog();
a.makeSound();  //this will print "woof woof!"

a = new Cat();
a.makeSound();  //this will print "meow meow!"
Interfaces

- An outline of behaviors to follow.
- Contains only **public, abstract** methods.
- The `implements` keyword. A child class **implements** an interface.
- Unlike inheritance, you *can* implement multiple interfaces.
Interfaces

```java
interface CanFly {
    void fly();
}

class Puffin implements CanFly, CanSwim {
    public void fly() {
        System.out.println("Hey I'm flying cuz I'm a puffin!");
    }
    public void swim() {
        System.out.println("Hey I'm swimming cuz I'm a puffin!");
    }
}

interface CanSwim {
    void swim();
}

class Penguin implements CanSwim {
    public void swim() {
        System.out.println("Hey I'm swimming cuz I'm a penguin!");
    }
}

class Pigeon implements CanFly {
    public void fly() {
        System.out.println("Hey I'm flying cuz I'm a pigeon!");
    }
}
```

multiple interfaces allowed
Abstract Classes vs Interfaces

- They are similar, but when to use which?
- If you’re just outlining behaviors, interface is fine.
- If you want to build off of existing behavior, abstract classes will do.
Encapsulation

- Data/information hiding
- Create getter and setter functions for properties.
- Protect your privacy!
- Allows user to be unaffected by implementation changes.
Encapsulation

- Recall the Bear class from the quiz:

```java
/* This is not good. Both height and weight are exposed and can be manipulated by anybody with access to a Bear object. */
class Bear {
    int height;
    int weight;
    String species;
    String color;
    void maul();
    void hibernate();
    void climb();
    void swim();
}

Bear smokey = new Bear();
smokey.weight = 400;

/* This is better. Height and weight are now read-only to the user. A user can no longer modify them directly. */
class Bear {
    private int height;
    private int weight;
    public int getHeight() { return height;}
    public int getWeight() { return weight;}
    String species;
    String color;
    void maul();
    void hibernate();
    void climb();
    void swim();
}

Bear smokey = new Bear();
int smokeyWeight = smokey.getWeight();
```
Composition

- A class that’s composed of objects of other classes within its definition.
- “Has-a” relationship, as opposed to inheritance which is “is-a”.
- Better than inheritance at adapting to code change.
Composition vs Inheritance

- Let’s revisit the Penguin/Puffin/Pigeon example.
- What if we added glide() to the CanFly interface and dive() to the CanSwim interface?
- We have to update the code for Penguin, Puffin, and Pigeon.
Composition vs Inheritance

- Puffin’s code doubles!

```java
interface CanFly
{
    void fly();
    void glide();  //new!
}

interface CanSwim
{
    void swim();
    void dive();  //new!
}

class Puffin implements CanFly, CanSwim
{
    public void fly()
    {
        System.out.println("Hey I'm flying cuz I'm a puffin!");
    }
    public void glide()
    {
        System.out.println("Hey I'm gliding cuz I'm a puffin!");
    }
    public void swim()
    {
        System.out.println("Hey I'm swimming cuz I'm a puffin!");
    }
    public void dive()
    {
        System.out.println("Hey I'm diving cuz I'm a puffin!");
    }
}
```
Composition vs Inheritance

- Instead of Puffin being a bird that can fly and can swim, how about a Puffin being a bird that has flying behavior and has swimming behavior?
- **Puffin is a flying and swimming bird** vs. **Puffin has flying and swimming capabilities**.
- If behavior details change, Puffin class does not change.
Composition vs Inheritance

- Puffin’s code stays the same.

```java
interface FlyAbility {
    void fly();
}

interface SwimAbility {
    void swim();
}

class Puffin {
    private SwimAbility swimming;
    private FlyAbility flying;

    void fly(FlyAbility ability) {
        flying = ability;
        flying.fly();
    }

    void swim(SwimAbility ability) {
        swimming = ability;
        swimming.swim();
    }
}

class PuffinFly implements FlyAbility {
    public void fly() {
        System.out.println("Hey I'm flying cuz I'm a puffin!");
    }
}

class PuffinGlide implements FlyAbility {
    public void fly() {
        System.out.println("Hey I'm gliding cuz I'm a puffin!");
    }
}

class PuffinSwim implements SwimAbility {
    public void swim() {
        System.out.println("Hey I'm swimming cuz I'm a puffin!");
    }
}

class PuffinDive implements SwimAbility {
    public void swim() {
        System.out.println("Hey I'm diving cuz I'm a puffin!");
    }
}
```
Data Structures and Algorithms

- Array
- ArrayList
- HashTable
- Stack
Array

- A linear sequence of items accessible by index.
- Pros: Fast random access.
- Cons: Fixed-size (i.e. cannot grow/shrink). Average search speed.

```java
String [] array = {"peter", "piper", "picked", "a", "peck"}; // int array [] works too
System.out.println(array[0]); //this would print "peter"
System.out.println(array[2]); //this would print "picked"
System.out.println(array[4]); //this would print "peck"
```
Array\List

- Dynamic version of Array.
- Pros: Fast random access.
- Cons: Overhead costs to grow/shrink. Still average search speed.

```java
import java.util.ArrayList;

ArrayList<String> arrayList = new ArrayList<String>(Arrays.asList("peter", "piper", "picked"));
arrayList.add("a");
arrayList.add("peck");
System.out.println(arrayList.get(0)); // this would print "peter"
System.out.println(arrayList.get(2)); // this would print "picked"
System.out.println(arrayList.get(4)); // this would print "peck"
System.out.println(arrayList); // this would print ":[peter, piper, picked, a, peck]"
```
Hashtable

- A table of **key-value pairs**. Look-up items using the key.
- **Pros**: Fast look-up speed.
- **Cons**: Can take up a lot of extra space.

```java
import java.util.Hashtable;

Hashtable<String, String> hometowns = new Hashtable<String, String>();
hometowns.put("Ten-Seng Guh", "Taipei, Taiwan");
hometowns.put("Stephen Curry", "Akron, Ohio");
hometowns.put("Sophia Guh", "Sleepy Hollow, New York");

System.out.println(hometowns.get("Ten-Seng Guh"));  //this will print "Taipei, Taiwan"
```
Stack

● A collection of items that are accessed in LIFO (last in, first out) order.
● Pros: Good for accessing latest items.
● Cons: Bad for accessing earliest items.

```java
import java.util.Stack;

Stack<String> shoppingCarts = new Stack<String>();
shoppingCarts.push("Cart 1");
shoppingCarts.push("Cart 2");
shoppingCarts.push("Cart 3");

String gimmeCart = shoppingCarts.pop();
System.out.println(gimmeCart); //this will print "Cart 3"
```
Homework 2

- Write a Blackjack program!
- Due July 14th (you have two weeks!)
- First think about your design:
- what data structure would you use to represent a deck of cards?
- How would you design the card class?