Computer Science

The science of using and processing **large amounts of information to automate useful tasks and learn about the world around us**. (using a computer)

OOP

Idea of OOP : Structure a program around the real-world concept of an object.

Class

CLASS: A type of data

OBJECT: One such piece of data with associated funtionality

Class Declaration

```
// Must be in file SimpleLocation.java
public class SimpleLocation
{
    // Member variables: data the objects need to store.
    // They are not local variables because they aren't declared inside a method..
    // Each object has its own value of member variable
    public double latitude;
    public double longitude;

     // Constructor: Method to create a new object. No return type.
    public SimpleLocation(double lat, double lon){
        this.latitude = lat;
        this.longitude = lon;
    }
```

```
    //Method: what the class can do
    public double distance(SimpleLocation other){
        ...
    }
}
```

"this" refers to the calling obkect.

Overloading Method

```
public class SimpleLocation
{
    public double latitude;
    public double longitude;
    // Default Constructor(no parameter). Overloading method.
    public SimpleLocation()
    {
        this.latitude = 32.9;
        this.longitude = -117.2;
    }
    public SimpleLocation(double lat, double lon)
    {
        this.latitude = lat;
        this.longitude = lon;
    }
}
```

Return type is NOT part of the method signature.Overloaded mathods CAN have different return types. Changing the return type is NOT ENOUGH for overloading. The parameter list must be different.

Public & Private

Public means can access from any class. (**for world use**)

Private means can access only from the class itself. (**helper methods**)

Rule of Thumb

Make the menber variables private. (methods either public or private)

Getter & Setters

Getter: gets the value of the member variable and returns it.

```
public class SimpleLocation
{
    private double latitude;
    // Getter. Users are not allowed to change the variable.
    public double getLatitude()
    {
        return this.latitude;
    }
}
```

```
public class LocationTester
{
    public static void main(String[] args)
    {
        System.out.println(lima.getLatitide());
    }
}
```

Setter: Takes in a value and changes the value. Sets the value of private menmer variable.

```
public class SimpleLocation
{
```

```
    private double latitude;

    public double getLatitude()
    {
        return this.latitude;
    }
    // Setter
    public void setLatitude(double lat)
    {
        this.latitude = lat;
    }
}
```

```
public void setLatitude(double lat)
{
    if (lat < -180 || lat > 180){
        System.out.println("Illegal value for latitude.")
    }
    else {
        this.latitude = lat;
    }
}
```

Static

static means that the variable or method marked as such is available at the class level. In other words, you don't need to create an instance of the class to access it.

```
public class Solution {
  int memberVariable = 0;

  void nonStaticMethod() {
    ++memberVariable;
    System.out.println("nonStaticMethod: " + memberVariable);
```

```
  }

  static void staticMethod() {
    // ++memberVariable; bug!
    System.out.println("staticMethod");
  }

  public static void main(String[] args) {
    staticMethod();
    // nonstaticMethod();  bug!
    Solution s = new Solution();
    s.nonStaticMethod();
  }
}
```

Memory Models

Trace Code : Reason about code

Temporarily store values

Primitive Types & Object Types

8 Primitive Types: boolean, byte, short, int, long, float, double, char

Object Types: Arrays and classes

```
SimpleLocation ucsd;
// new: allocate space to store the object in The Heap.
// Place a reference to the object (the address/location in the heap).
ucsd = new SimpleLocation(32.9, -117.2);
```

```
public class LocationTester
{
    public static void main(String[] args)
    {
        SimpleLocation loc1 =
            new SimpleLocation(39.9, 116.4);
        SimpleLocation loc2 =
            new SimpleLocation(55.8, 37.6);
        // Copy the address of loc2!
        loc 1 = loc 2;
        loc1.lat = -8.3;
        System.out.println(loc2.lat + "," + loc2.lon);
        // The output is "-8.3, 37.6".
    }
}
```

Variable Scope

The scope of a variable is the area where it is defined to have a value.

Pass by Value

Member variables are declared outside any method. Can be used in the whole class.

Constructor's scope consits of constructor's parameters, and it desaperas when the method ends.

Quiz 2

```
//Because Java is pass by value, a copy of that value is passed into the method.
//Changing the copy of the primitive type will have no impact on the parameter.
//This is a tricky question, because methods incrementA and incrementB have no real purpose.

public class MyClass
```

```
{
  private int a;
  public double b;

  public MyClass(int first, double second)
  {
    this.a = first;
    this.b = second;
  }

  public static void incrementBoth(MyClass c1) {
    c1.a = c1.a + 1;
    c1.b = c1.b + 1.0;
  }

  // new method
  public static void incrementA(int first)
  {
    first = first + 1;
  }

  // new method
  public static void incrementB(double second)
  {
    second = second + 1.0;
  }

  public static void main(String[] args)
  {
    MyClass c1 = new MyClass(10, 20.5);
    MyClass c2 = new MyClass(10, 31.5);
    // different code below
    incrementA(c2.a);
    incrementB(c2.b);
    System.out.println(c2.a + ", "+ c2.b);
  }
}
// Returns "10, 31.5".
```

Graphical User Interfaces

Class PApplet

GUI data type

Inheritence (keyword : extends)

E.g : Fully written Person class now needs to handle: 1. Students 2.Faculty

Consistency? Storing all objects in one data structure?

GOALS:

- Keep common behavior in one class
- Split different behavior into separate classes
- Keep all of the objects in a single data structure

```
Class A {
  int a1;
  int a2;
}

class B {
  float b1;
  float b2;
}

class C {
  A a;
  B b;
}
```

```
Class c = new Class();
// If you want to access a1:
println(c.a.a1);
```

terminology

```
// base/ super class
public class Person
{
    private String name;
    public String getName
    {
        return name;
    }
}

// derived/ sub Class
public class Student extends Person
{
    // private String name; (inherited, hidden variable)
    private String GPA;
    public String getGPA
    {
        return name;
    }
}
```

What is inherited?

- Public instance variables
- Public mathods
- Private instance variavles

Private vars can be accessed ONLY through public methods! - have to use getters and setters to access name.

Design an inheritance hierarchy - UML Diagram

Common Code - > Diverging Code

Reference vs. Object Type

Student inherited from person:

```
Person p = new Student();
// A Student "is-a" person;
```

so : A Person array CAN store Student and Faculty objects

```
// in main
Person[] p = new Person[3];
p[0] = new Person();
p[1] = new Student();
p[2] = new Faculty();
```

Quiz

```
public class Person {
    private String name;
    public String getName(){
        return name;
    }
}

public class Student extends Person {
```

```
    private int id;
    public int getID(){
        return id;
    }
}

public class Faculty extends Person {
    private String id;
    public String getID(){
        return id;
    }
}

Student s = new Student();
Person p = new Person();
Person q = new Person();
Faculty f = new Faculty();
Object o = new Faculty();
```

- `String n = s.getName();`
- `p = s;` Person reference refer to student object.(Is a student a person?)
- Error: `int m = p.get ID` Compiler dosen't know object type. Cast: `int m = ((Student)p).get ID`
- Error: `f = q` Is a person falcuty? Cannot cast person to faculty.
- `o = s` **Everything in Java is an Object**
-

Visibility Modifiers

Rule of Thumb: always use either public or private

- public
  - can access from **any class**
- protected
  - can access from **same class**

    - can access from **same package**
    - can access from **any subclass**
- package(defualt)
    - can access from **same class**
    - can access from **same package**
- private
    - can access from **same class**

Object Creation

```
Student s = new Student();
```

- "new" allocates space (is an operator)
- "Student()" is passed to the constructor for initialization
- Objects are created from inside out. Object - > Person - > Student

Complier Rules for Class Construction

- Your Code
- Java Compiler (Processes code and inserts new commands)
- bytecode (run in JVM)

Compiler's rules:

1. No superclass? Compiler inserts: `extends Object`
2. No constructor? Java gives you one for you.
3. First line must be `this(args..)` or `super(args...)`, Otherwise Java inserts: `super();`

```
public class Person extends Object {
    private String name;
    public Person(){
```

```
        super();
    }
}

public class Student extends Person {
    public Student(){
        super();
    }
}
```

Variable Initialization in a Class Hierarchy

```
// How to initialize name without a public setter?
public class Person extends Object {
    private String name;

    public Person( String n){
        super();
        this.name = n;
    }
}
public class Student extends Person {
    // Defalut constructor;
    public Student(){
        // Use our same class constructor
        this("student"); // or super("student");
    }

    public Student( String n){
        super(n);
        // cannot use 'this.name = n;' because name is private;
    }
}
```

Method Overriding

- **Overloading** : *Same class* has same method name with *different parameters*.
- **Overriding** : *Subclass* has same method name with the *same parameters* as the superclass.

```
package week4;

public class Person {
  private String name;

  public Person(String n) {
    name = n;
  }

  public String getName() {
    return name;
  }

  @Override
  public String toString() {
    return getName();
  }

  public static void main(String[] args) {
    Person p = new Person("Tim");
    System.out.println(p); // p.toString() is needless.

    Student s = new Student("Cara", 1234);
    System.out.println(s); // output: 1234:Cara

    // polymorphism
    Person s2 = new Student("Cara", 1234);
    System.out.println(s2);// output: 1234:Cara
  }
}

class Student extends Person {
  private int studentID;

  public Student(String n, int id) {
```

```
    super(n);
    studentID = id;
  }

  public int getSID() {
    return studentID;
  }

  @Override
  public String toString() {
    return this.getSID() + ":" + super.getName(); // In case person name changes
  }
}
```

Polymorphism

Superclass reference to subclass object

```
Person s = new Student("Cara", 1234);
System.out.println(s);
// toString() method depends on dynamic type
```

Rules to Follow for Polymorphism

Compiler Time Rules

- Compiler ONLY knows *reference type*
- Can only look in reference type class for method
- outputs a method signature (which will be executed at runtime)

Run Time Rules

- Follow exact *runtime type* of object to find method
- Must match compile time method signature to appropriate method in actual object's clsss

Casting Objects

- **Widening**:Automatic type promotion. `Superclass ref = new Subclass();`
- **Narrowing**: Explicit casting. `Subclass ref = (Subclass) superRef;`

Runtime type Check

Provides runtime check of **is-a** relationship

```
if (s instanceof Student) {
    // only exectutes if s is-a Student
    // at runtime ((Student)s).getSID();
}
```

Abstract Classes and Interfaces

- Force subclasses to have this method
- Stop having actual Person objects
- Keep having Person references
- Retain common Person code

Abstract

- Can make any class abstract with keyword: `public abstract class Person{ //Cannot create objects of this type`
- Class *must* be abstract if any methods are: `public abstract void mothlyStatement(){ //concrete subclasses must override this method`

Implementation & Interface

Abstract classes offer inheritance of both!

- **Implementation**: instance variables and methods which define common behaviour. *(Retain common code)*
- **Interface**: mathod signatures which define required behaviors

Interfaces

- Interfaces only define required methods
- Classes can inherit from multiple interfaces

Inheritence

```
class Base {
  public void foo() {
    System.out.println("base.foo");
  }
}

class Derived1 extends Base {
  public void bar() {
    System.out.println("Derived1.bar");
  }
}

class Derived2 extends Base {
  public void bar() {
    System.out.println("Derived2.bar");
  }
}

public class Inheritance {
        public static void main(String[] args) {
          Base b = new Base();
          b.foo();

          Derived1 d1 = new Derived1();
```

```
            d1.bar();

            Derived2 d2 = new Derived2();
        d2.bar();
          }
}
```

Interface

```
interface Printable {
  public void print();
}

interface Moveable {
  public void move(double a, double b);
}

abstract class MachineBase implements Printable, Moveable {
  double x_, y_;
}

// Java does NOT allow multiple inheritance.
class Machine1 extends MachineBase {
  public void print() {
    System.out.println("Machine1");
  }
  public void move(double x, double y) {
    x_ = x;
    y_ = y;
  }
}

class Machine2 extends MachineBase implements Printable, Moveable {
  public void print() {
    System.out.println("Machine2");
  }
```

```
  public void move(double radius, double theta) {
    x_ = radius * Math.sin(theta);
    y_ = radius * Math.cos(theta);
  }
}

public class Interface {
  public static void main(String[] args) {
    MachineBase[] machines = {
      new Machine1(),
      new Machine2()
    };
    for (MachineBase m: machines) {
      m.print();
    }
  }
}
```

Abstract Class

```
// Abstract method:
//   - a method without body
//   - a method needed to be implemented by other class (either `implements` or `extends`)

interface I {
  // public as default
  // Foo is a required method, which means it require the class who `implements`
  // interface I must implement the methods defined in I.
  void foo();

  default void bar() {
    System.out.println("This is Interface I.");
  }
}

class A implements I {
```

```
  public void foo() {

  }
}

abstract class B {
  // Foo is a required method, which means it require the class who `extends`
  // this class must implement the abstract methods defined here.
  abstract void foo();

  // Common behavior: when you want to access (read or write) its member variable
  void printA() {
    System.out.println("This is base class B");
    System.out.println(a);
  }

  int a = 20;
}

class C extends B {
  void foo() {
  }
}

class D extends B {
  void foo() {
  }
}

// The compiler always has to `resolve` something.
public class AbstractClassAndInterface {

  public static void main(String[] args) {
    // // bug
    // new Interface();
    // // bug
    // new B();
  }
}
```

```
abstract public class AbstractPerson {
  // defined a required method
  abstract String getPassword();

  public static void main(String[] args) {
    AbstractPerson[] ps = { new StudentFoo(), new Faculty() };
    // 1: base class has defined this abstract method
    // 2: base class cannot be created, while the derived class can be.
    // 3: derived class must has implemented this method
    for (AbstractPerson p : ps) {
      System.out.println(p.getPassword());
    }
  }
}

class StudentFoo extends AbstractPerson {
  String getPassword() {
    return "student pwd";
  }
}

class Faculty extends AbstractPerson {
  String getPassword() {
    return "faculty pwd";
  }
}
```

Event-Driven Programming

Procedural

Code exectution follows predictable sequence, based on control logic and program state.

Event

Listener - react to user input

Override the keyPress method from supercalss PApplet

Customize user interface using events

Button

- Draw buttons (Insert codes in draw();)
- Add button functionality : Overriding methods: mousePressed(), mouseClicked(), **mouseReleased()**...

Listener Hierarchy

- PApplet implements tow interfaces: MouseListener & KeyListener
- We overeide methods

Searching

Linear Search

```
// need to perform n comparisons
public static String findAirportCode(String toFind,
                ArrayList<Airport> airports)
{
        int index = 0;
        while (index < airports.size()) {
                if (toFind.equals(airports.get(index).getCity())) {
                        return airports.get(index).getCode3();
                }
                index++;
```

```
        }
        return null;
}
```

Binary Search

List must be sorted.

```
// Log base two n times
public static String findAirportCodeBS(String toFind, ArrayList<Airport> airports)    {
        int low = 0;
        int high = airports.size()-1;
        int mid;
        while (low <= high) {
                // possible overflow if mid = (high + low) / 2;
                mid = low + ((high-low)/2);
                int compare = toFind.compareTo(airports.get(mid).getCity());
                if (compare < 0) {
                        high = mid - 1;
                }
                else if (compare > 0) {
                        low = mid+1;
                }
                else return (airports.get(mid)).getCode3();
        }
        return null;
}
```

Sorting

Duplicates?

Selection Sort

For each position i from 0 to length - 2.

Invariant Property

```
public static void selectionSort (int[] vals) {
    for (int i = 0; i < vals.length - 1; i++) {
        indexMin = i;
        for (int j = i + 1; j < vals.length; j++){
            if (vals[j] < vals[indexMin]){
                indexMin = j;
            }
        }
        swap(vals, indexMin, i);
    }
}
```

Insertion Sort - relativly

```
public static void mysterySort(int[] vals){
    int currInd;
    for(int pos = 1; pos < vals.length; pos++){
        currInd = pos;
        while (currInd > 0 && vals[currInd] < vals[currInd - 1]){
            swap(vals, currInd, currInd - 1);
            currInd = currInd - 1;
        }

    }
}
```

Bulit-in Sort

Merge-sort - see "Data Structures and performances"

```
import java.util.*;
// Integer numbers
Collections.sort(nums);
```

Comparable Interface

```
public class Airport implements Comparable<Airport> {
    public int compareTo(Airport other) {
        return getCity().compareTo(other.getCity());
    }
    public static void main (String[] args) {
        ArrayList<Airport> airports = new ArrayList();
        ...
        airporys.add(...);
        ...
        Collections.sort(airports);
    }
}
```