Object-Oriented Design

Summary: The most important things to remember about Object-Oriented Design are:

- Divide up your problem and its solution into parts (=classes & objects).
- When you divide, make the interactions (method calls and field references) as simple and easy to understand as possible;
- Make the interface follow KISS -- **provide as few public methods as possible**;
- Use **Information Hiding**: Hide as much about your implementation as you possibly can. If you are not sure whether to make something public or private, **make it private**;

The advantages of information hiding are:

- Your code is easier to **understand**, and hence to **use**, and **reuse**;
- Users can't get used to "back-door" ad-hoc features of your code;
- By separating the (simple) **behavior** of your system from the messy details of its **implementation**, you can **change the actual implementation** any time you want---as long as it behaves the same, this is a huge advantage for **maintenance and reuse**.
Over the years, system designers have defined a number of **standard design patterns** for the parts and interactions of a program. The most basic pattern is a single file implementing a simple task:

**Stand-Alone Program:**

```java
public class PrintDay {
    public static void main(String[] args) {
        int day = 1;
        int month = 3;
        int year = 1970;

        int y0 = year - (14 - month) / 12;
        int x = y0 + y0/4 - y0/100 + y0/400;
        int m0 = month + 12 * ((14 - month) / 12) % 7;
        int d0 = (day + x + (31 * m0)/12) % 30;

        String[] dayOfWeek = {"Sunday", "Mo", "Tu", "We", "Th", "Fr", "Sa"};
        System.out.println("Unix\'s birthday is on " + dayOfWeek[d0]);
    }
}
```

**Rules:**
- Everything is static;
- `main(...)` is public;
- ALL OTHER members are private;
- Uses no libraries!

A “stand-alone program” is not very useful! More common is a program which uses the standard Java libraries as a **Client** to accomplish some task:

**Client: Histogram**

```java
import java.util.Scanner;
public class Histogram {
    private static final int MAX NUMBERS = 20;

    private static void printHeading() {
        System.out.println("This program will print a histogram of the numbers entered by the user. Enter up to 20 numbers.");
    }

    public static void main(String[] args) {
        Scanner userInput = new Scanner(System.in);
        printHeading();
        System.out.println("Enter up to 20 numbers:");
        System.out.println("Enter -1 to finish.");
        int number;
        while ((number = userInput.nextInt()) != -1) {
            if (number < 0) {
                break;
            }
            System.out.println("Number is: " + number);
            histogram[number-1] ++;
        }
    }
}
```

**Client Rules:**
- Everything is static;
- `main(...)` is public & controls execution;
- ALL OTHER members are private;
- Uses standard Java libraries.
A client may use standard Java libraries (Scanner, Math, String, Character, ...) or may use a static library written by the user:

**Client:** HW03Client

**Static Library:** BigInt

**Static Library Rules:**
- Everything is static;
- Interface is small & public;
- Implementation is private;
- Stores no local data;
- May itself use libraries;
- Static main used to store testing code.

**Client Rules:**
- Everything is static;
- Main(...) is public & controls execution;
- ALL OTHER members are private;
- May use standard Java libraries;
- May use programmer-defined static libraries;
- Does not define any objects.

A client may also use one or more Objects it creates dynamically to hold data and related algorithms, called an Abstract Data Type:

**Static Library:** BigInt

**Static Library Rules:**
- Everything is static;
- Interface is small & public;
- Implementation is private;
- Stores no local data;
- May itself use libraries;
- Static main used to store testing code.

**Client:** HW03Client

**ADT:** S

**Abstract Data Type Rules:**
- Interface is small & public;
- Implementation is private;
- Stores data and related algorithms;
- May itself use libraries;
- Main used to store testing code, and is only static member.
Object-Oriented Design: Design Patterns

Sometimes this is called the **Client/Server Model:**

Client: Controls execution of whole program.

Servers (store data with associated algorithms used by client):

Libraries: Store code used by client.

---

Abstract Data Types: The Stack ADT

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

Stack Interface (informal):

- **void Push(int n)** - Put integer n on top of the stack
- **int Pop()** - Remove top integer and return it
- **int Peek()** - Return the top integer without removing it
- **int size()** - Return the number of integers in the stack
- **boolean isEmpty()** - Returns true iff stack has no members

---
Abstract Data Types: The Stack ADT

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

**Stack Interface:**

```java
void Push(int n)
ing Pop()
ing Peek()
ing size()
boolean isEmpty()
```

```java
push(5);  // 5
push(7);  // 7
```

5

9
Abstract Data Types: The Stack ADT

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

Stack Interface:

```java
void Push(int n)
int Pop()
int Peek()
int size()
boolean isEmpty()
```

push( 5 );
push( 7 );
push( 2 );

5
7
2

Abstract Data Types: The Stack ADT

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

Stack Interface:

```java
void Push(int n)
int Pop()
int Peek()
int size()
boolean isEmpty()
```

push( 5 );
push( 7 );
push( 2 );
int n = pop();

n: 2

2
7
5
Abstract Data Types: The Stack ADT

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

**Stack Interface:**

```java
void Push(int n)
int Pop()
int Peek()
int size()
boolean isEmpty()
```

push(5);
push(7);
push(2);
int n = pop();
int m = pop();

5

n: 2
m: 7

5

Abstract Data Types: The Stack ADT

The **Stack ADT** is perhaps the simplest: it defines how a pile of objects works: you can only modify the top of the stack!

**Stack Interface:**

```java
void Push(int n)
int Pop()
int Peek()
int size()
boolean isEmpty()
```

push(5);
push(7);
push(2);
int n = pop();
int m = pop();
int i = pop();

5

n: 2
m: 7
i: 5
Abstract Data Types: The Stack ADT

Applications of Stacks:

- Reversing an array or a String
- Keeping track of nested or recursive structure
- Parenthesis Matching
- Evaluating an arithmetic expression
- Run-time Stack to keep track of method/function calls

[Examples on Board]

Problems with Stacks:

- Underflow: Trying to pop() or peek() and empty stack! Solution: check if empty before doing a peek or pop!
- Overflow: Pushing too many numbers and causing an ArrayIndexOutOfBoundsException! Solution: Array Resizing....
Reference types: Data in Computer Memory

To understand the notion of references (also called pointers), we need to understand how computer memory works to organize data:

Computer instructions say things like:

“Put a 3 in location 8:"

RAM[8] = 3;

“Add the numbers in locations 8 and 9 and put the sum in location 2:"


This is why arrays are so common and so efficient: RAM is just a big array!

Access time = about $10^{-7}$ secs

When you create variables in Java (or any programming language), these are “nicknames” or shortcut ways of referring to locations in RAM:

These “shortcut” names for primitive types can not change during execution.

When we draw our diagrams of variables, we are really just giving a shortcut view of RAM without the addresses:
NOW we can change the “meaning” of the reference variable by assigning it a new location; in fact, \texttt{new} returns the new location, which is stored in the reference variable as its “value.”
Reference types: Objects/Classes in Computer Memory

Now we can change the “meaning” of the reference variable by assigning it a new location; in fact, new returns the new location, which is stored in the reference variable as its “value.”

Old objects are “garbage” and the memory will be reclaimed by the “garbage collection” and reused.

Reference Types: String

We have seen two different reference types so far in this course:

The first is Strings:

```java
public class Strings{
    public static void main(String[] args) {
        String s = "hi there";
        String t = new String( "hi there" );
        String u = "Hi There!";

        System.out.println( s.equals( t ) );
        System.out.println( s.equals( u ) );
        System.out.println( s == t );
        System.out.println( s == u );
    }
}
```
Reference Types: String

We have seen two different reference types so far in this course:

**The first is Strings:**

```java
public class Strings {
    public static void main(String[] args) {
        String s = "hi there";
        String t = new String("hi there");
        String u = "Hi There";

        System.out.println(s.equals(t));  // equals checks for same structure;
        System.out.println(s == t);      // == checks for same reference (pointing to same location).
    }
}
```

Reference Types: String

We have seen two different reference types so far in this course:

**The second is arrays:**

Let's look at how to solve the problem of stack overflow, using array resizing:

```java
// replace S by array twice as big, but with same elements
private void resize() {
    int[] T = new int[S.length * 2];
    for (int i = 0; i < S.length; ++i) {
        T[i] = S[i];
    }
    S = T;
}
```