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:**

```
static main(...) {
    // Code goes here
}
```

**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

```
static main(...) {
    // Code goes here
}
```

**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

**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.
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!

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

Stack Interface:

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

5

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:

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

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

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

2
7
5

n: 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()
```

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

- `n`: 2
- `m`: 7
- `i`: 5

```
5
```
Abstract Data Types: The Stack ADT

**Applications of Stacks:**

- Reversing an array or a String
- Keeping track of calls waiting
- 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....
To understand the notion of references (also called pointers), we need to understand how computer memory works to organize data:

**RAM:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>13</td>
<td>23</td>
<td>-34</td>
<td>232</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>10</td>
<td>-78</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>-1</td>
<td>?</td>
</tr>
</tbody>
</table>

Computer instructions say things like:

"Put a 3 in location 8:"

\[ \text{RAM}[8] = 3; \]

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

\[ \text{RAM}[2] = \text{RAM}[8] + \text{RAM}[9] \]

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:

**RAM:**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>13</td>
<td>23</td>
<td>-34</td>
<td>232</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>10</td>
<td>-78</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>-1</td>
<td>?</td>
</tr>
</tbody>
</table>

int x;  // same as RAM[8]
int y;  // same as RAM[9]
int z;  // same as RAM[2]

// now the previous computation
// would be

\[ x = 3; \]
\[ y = 10; \]
\[ z = x + y; \]

When we draw our diagrams of variables, we are really just giving a shortcut view of RAM without the addresses:

x: 3
BUT Reference Types (arrays, Strings, objects – anything you can use the word \texttt{new} to create new instances of) are \textbf{references} or \textbf{pointers} to their values: they store the location of the value, not the value itself.

```
int x;
int y;
int z;
```

```
int [] A = { 11, 3, 4 };  
Point P = new Point(5, -1);
```

```
A[0]: 11
A[1]: 3
A[2]: 4
```

```
P = new Point(2, 3);
```

```
P = new Point(2, 3);
```

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."

```
int [] A = { 11, 3, 4 };  
Point P = new Point(5, -1);
```

```
A = new int[2];  
P = new Point(2, 3);
```

```
P = new Point(2, 3);
```

```
P = new Point(2, 3);
```

```
P = new Point(2, 3);
```

```
P = new Point(2, 3);
```
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 ) );  // true
        System.out.println( s.equals( u ) );  // false
        System.out.println( s == t );         // false
        System.out.println( s == u );         // false
    }
}
```

equals checks for same structure;

== checks for same reference (pointing to same location).

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;
}
```
The **Queue ADT** is a simple variant of a stack which makes a simple change which in fact changes everything: instead of moving items in and out of the same "end" of the list, as in a stack:

Instead you use different ends of the list:

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office!), I'll only give a brief example:
Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

enqueue(5);

enqueue(7);

Enqueue 5 Dequeue

Enqueue 7 5 Dequeue
Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

enqueue(5);
enqueue(7);
enqueue(2);

```c
int k = dequeue();
```

k = 5
Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

enqueue(5);
enqueue(7);
enqueue(2);
int k = dequeue();
enqueue(8);

```
enqueue(5);
enqueue(7);
enqueue(2);
int k = dequeue();
enqueue(8);
```

Enqueue             Dequeue

```
8        7
2
k = 5
```

Queue ADT

This means that instead of reversing the order of the items, as with a stack, they remain in the same order; since you have stood in lines many times at Starbucks (or outside my office), I'll only give a brief example:

enqueue(5);
enqueue(7);
enqueue(2);
int k = dequeue();
enqueue(8);
enqueue( dequeue() )

```
enqueue(5);
enqueue(7);
enqueue(2);
int k = dequeue();
enqueue(8);
enqueue( dequeue() )
```

Enqueue             Dequeue

```
7        8        2
k = 5
```
Queue ADT

Queues occur all the time, in real life:

And in computer systems:

In fact, anywhere where one service is desired by many, and must be fairly distributed... there is a whole branch of math called "queueing theory" which you will learn about in CS 237 and CS 350.....

Array-based Implementation of Queues

A Queue for integers could be defined by the following interface:

```java
public interface Queueable {
    void enqueue(int key); // insert the key at the end of the queue
    int dequeue();         // remove the key at front of the queue
    boolean isEmpty();     // returns true if queue is empty
    int size();            // returns number of integers in queue
}
```

How to implement this with arrays?
Array-based Implementation of Integer Queues

To implement an array-based queue for ints, here is the first thing you might think of.....

```
void enqueue(int k) {
    A[next] = k;
    ++next;
}
int dequeue() {
    int temp = A[front];
    ++front;
    return temp;
}
int size() {
    return (next – front);
}
```

But there is an obvious problem, and not so trivial..... running off the end of the array!

```
void enqueue(int k) {
    if(size() != A.length) {
        A[next] = k;
        ++next;
    }
}
int dequeue() {
    int temp = A[front];
    ++front;
    return temp;
}
Boolean isEmpty() {
    return (size == 0);
}
```
Array-based Implementation of Integer Queues

What solutions could we come up with for this problem?

Well, there are several:

Bad: Resize the array so you don’t run off the end. But then your array grows and grows and grows!

Good: Each time you dequeue, shift all the data over (similarly with how a queue is managed in Starbucks: when the person at the head of the line leaves, everyone moves up!). A natural solution, but if the queue is very large, each dequeue takes a long time, since you have to touch every data item and move it.

Best: Consider the array to be in a circle, with each end “glued” together, so that you never run off the array....