Java Classes and OO Programming

- Unlike most other programming languages, all variables and methods/functions must be contained in a class.

- The idea of a class is that it represents a single abstraction of some kind.

- Variables represent the state of the abstraction, and methods represent operations or functions of the abstraction.

- First, we’ll consider classes with only static members (variables and methods) using a stack example.

Your First ADT: The Stack

- A stack is an abstract data type, perhaps the best known in all of computer science.

- Conceptually, it is a list which has (at least) two operations:
  - push — add an elt onto the top of the stack.
  - pop — remove the most recently added elt.

- Enforces “LIFO” (last in is the first out) policy on additions and deletions.

  - Cannot add elts to or delete elts from anywhere but the “end” (top) of the stack.

- Other common operations:
  - peek — return (a copy of?) the top elt without removing it.
  - size — return # of elts (aka length, etc.)
  - isEmpty — return true iff there are no elts in the stack, return false otherwise.

// Note: This class implements a stack of Strings only.
class SingleStack {
    public static final int MaxSize = 100;
    private static int top = -1;
    private static String[] store = new String[MaxSize];

    public static void push (String newVal) {
        if (top < MaxSize -1) {
            top++;
            store[top]=newVal;
        } else {
            System.err.println("Oops, stack full.");
        }
    }

    public static String pop () {
        String ans = "";
        if (top >= 0) {
            ans = store[top];
            top--;
        } else {
            System.err.println("Oops, stack empty.");
        }
        return ans;
    }

    public static int size () {
        return top+1;
    }
}

class testSingleStack {
    private static String testCourseName = "CS211";

    public static void main (String [] args) {
        SingleStack.push ("CS100");
        SingleStack.push (testCourseName);
        String s = SingleStack.pop();
        System.out.println("popped: "+s);
        SingleStack.push ("CS314");
        s = SingleStack.pop();
        System.out.println("popped: "+s);
        s = SingleStack.pop();
        System.out.println("popped: "+s);
        // The stack should be empty now
        s = SingleStack.pop();
    }
}

Output is:

```
popped: CS211
popped: CS314
popped: CS100
```

// Sorry, stack is empty.

Problems with this Approach:

- There is only one “stack”; what if we want several?
Purposes of a Class Definition

Class definitions perform two independent functions:

1. **Class definition as module**
   - *i.e.*, class definition enforces information hiding.
   - Make internal details hidden (*e.g.*, private); only interface that represents the abstraction is available to clients (*e.g.*, public).\(^a\)
   - If all class members are declared as `static`, then class definitions essentially represent only walls and bridges, defining allowed component interactions within the system.

2. **Class definition as template** ("cookie cutter")
   - Each class defines a *kind* of entity.
   - Can create new instances as needed (using `new`).
   - Each instance has personal, distinct copy of all non-static variables.

\(^a\)Different clients may have different access permissions; more later

```java
// This class implements a stack of Objects, which could be Strings, Integers, etc. Note: Use java.util.Stack instead of this class in real life.
class stack {
    // The maximum possible size of the stack.
    public static final int MaxSize = 100;
    // The array index of the last active element in the stack. A value of -1 means stack is currently empty.
    private int top = -1;
    // The stack elements, stored in an array.
    private Object[] store = new Object[MaxSize];
    // Keep track of the number of valid pushes done to all stacks.
    private static int numValidPushes = 0;
    // Add a new element to the end of the stack (if there’s room left).
    public void push(Object newVal) {
        if (top < MaxSize - 1) {
            top ++;
            store[top] = newVal;
            numValidPushes ++;
        } else {
            System.err.println("Sorry, stack full.");
        }
    }
    // Remove the top element of the stack and return it; (check that the stack isn’t empty!)
    public Object pop() {
        Object ans = "";
        if (top >= 0) {
            ans = store[top];
            top--;
        } else {
            System.err.println("Sorry, stack empty.");
        }
        return ans;
    }
    // Return how many elts are currently in the stack.
    public int size() {
        return top + 1;
    }
    // Report number of valid pushes so far;
    public static int getNumValidPushes() {
        return numValidPushes;
    }
    // Print the stack from top to bottom.
    // (This is not a standard function of stacks.)
    public void print() {
        int i;
        for (i = top; i >= 0; i--) {
            System.out.println(store[i]);
        }
    }
}
```

```java
class testStack {
    private static String testCourseName = "CS211";
    private static String testGrade = "A-";
    public static void main (String[] args) {
        stack courseStack = new stack();
        stack gradeStack = new stack();
        courseStack.push("CS100");
        courseStack.push(testCourseName);
        gradeStack.push("B+");
        // Treat the Object as a String.
        String s = (String) courseStack.pop();
        System.out.println("popped :" + s);
        courseStack.push("CS314");
        gradeStack.push(testGrade);
        s = (String) courseStack.pop();
        System.out.println("popped :" + s);
        System.out.println("popped :" + s);
        courseStack.push(s);
        // Print the stacks:
        System.out.println("Here’s the course stack:");
        courseStack.print();
        System.out.println("Here’s the grade stack:");
        gradeStack.print();
        // Print out how many valid pushes so far.
        int howMany = stack.getNumValidPushes();
        System.out.println("Number of valid pushes: " + howMany);
    }
}
```
**static vs. Instance Variables**

- A variable declared as `static` means there is only ever one of them, period.
  - `static` variables are sometimes called *class variables*, because there is ever only one created per class.
  - You do not get a new one with each instance; there is only one, period.
- A non-static variable is called an *instance* variable.
  - When you instantiate from a class, you get a new copy of each (i.e., instance) variable.

  *i.e., instance variable mean there is one copy per instance of the class.*

**static vs. Instance Methods**

- A method/function declared as `static` means that it cannot directly access any instance variables of the class or call any non-static methods.
  - `static` functions may access static variables directly.
  - `static` functions may create new instances of the class and manipulate the instance variable/methods of those instances.
    (Often, main functions create instances of the same class to do some testing.)
- Non-static (i.e., instance) methods may access both static and instance variables.

**Constructors**

- A *constructor* is a special kind of function, called when an object is created/instantiated.
  - The constructor details what needs to be done to set up an instance of the class.
  - Called once per lifetime of an object via a call to new.
- You can tell a constructor from a “normal” method, as the constructor:
  1. Looks like a function definition, but the name is the same as the class.
  2. Doesn’t have a return type.
- You may define several constructors that differ in # and kind of parameters.
  - Only one constructor will be called on any object, ’tho.
  - This allows the client (i.e., the one who is instantiating) some flexibility is how to set up the object.

```java
class stack {
  // The maximum possible size of the stack.
  private int maxSize;

  // The default value for maxSize.
  public static final int DefaultMaxSize = 100;

  // The index of the last active element in the // stack. A value of -1 means stack is empty.
  private int top = -1;

  // The stack elements, stored in an array.
  private Object [] store;

  // Keep track of the number of valid pushes done // to all stacks.
  private static int numValidPushes = 0;

  // Create a stack of at most DefaultMaxSize elts.
  public stack () {
    maxSize = DefaultMaxSize;
    store = new Object [maxSize];
  }

  // Create a stack of at most desiredMaxSize elts.
  public stack (int desiredMaxSize) {
    maxSize = desiredMaxSize;
    store = new Object [maxSize];
  }
  // rest of class "stack" same as before
```
This is called **method overloading**.

- Same method name has multiple definitions.
- Each method definition has a different parameter list.
- System can figure out which definition to use by looking at how it was called.

```java
stack s1, s2;
s1 = new stack();
s2 = new stack (25);
```

- Can overload all methods, not just constructors.
- Related idea that is illegal in Java: **operator overloading**
  - C++ allows programmers to redefine what operators such as “+” and “=” mean.
  - This have proven confusing in practise; source of many very non-obvious errors in big systems.
  - Java has a little operator overloading built in, (e.g., s1+s2 means concatenate if args are Strings, add if args are numbers) BUT programmers may not redefine operators.

```
stack s1, s2, s3;
s1 = new stack();
s2 = new stack (25);
s3 = s1;
s2 = s3;
```

- s1, s2, and s3 are references to objects
- Instances are not created until `new` is called.
- To determine which constructor is used, match parameters!
- Arrays are a bit different; to create an array, put number of desired elements in `square` brackets:

```java
String [] nameList = new String [15];
```

- If you try to use s3 without instantiating to it or assigning it to point to an existing stack, you will get a `NullPointerException` thrown at you and your program will die.

### Deleting Objects?

- Facts of life:
  1. Each object takes up memory.
  2. Amount of available memory is finite.
  3. Often, we reach a point where we no longer need a particular object; want system to reclaim the storage for possible reuse later.
- In some languages, when you know you are done with an object, you must tell the system explicitly:

```cpp
// This is C++ code, not Java
Stack *s1;
s1 = new Stack (15);
// OK, I don’t need this object anymore.
delete s1;
// Carry on ...
```

The C++ philosophy on storage management is:

- **DIY — Do it yourself!**
DIY approach:

- Extra burden on programmer; keep track of when you are done with an object and explicitly say so.
- In C++, this can be unsafe!
  (Other OO languages practise safe DIY.)

Java approach is simpler:

- System automatically does garbage collection (GC) when it thinks the time is right.
- User never has to worry about freeing up storage; “garbage” objects will eventually be reclaimed by system.

GC is an old idea. There are several approaches; Java uses “mark and sweep”:

- Mark all objects “reachable” (by chains of references) from all active objects. Sweep up all objects that were not marked into the recycling bin.

Overview of Object-Oriented Ideas

Levels of OO-ness in programming languages:

1. Language supports modularity:
   
   - Information-hiding: interface vs. implementation distinction, prevent unauthorized access to internals

Advice to Java programmers:

- Class is a wrapper; some things are hidden, some are visible by clients.
- Make variables and methods as private as possible by default.
- Never make variables public.
- Make abstract “buttons” public; consider carefully what the interface should look like.
  - Design for the abstraction. Can clients understand it without looking at the rest of the code?

2. Language is object based:
   
   - Can define classes and instantiate from them.
   - Class is a cookie cutter; can stamp out as many cookies as you like.
   - Each instance has private state (i.e., its own copies of instance variable).

3. Language is fully object oriented:
   
   - It supports most of the following ideas:
     - Inheritance
     - Interfaces or multiple inheritance
     - Polymorphism and overloading
     - Genericity
     - Dynamic binding

Garbage Collection

- In general, GC is a good idea.

- Some practicality issues, ’tho:
  
  - “Bursty” performance possible.
    - [Java actually uses separate thread for GC; don’t worry about what this means.]
  
  - Less efficient, in general, than well-tuned DIY systems.
  
  - GC must be implemented correctly or else!
Java Variable/Method Visibility Levels

```java
class C {
    private int i;
    int j;
    protected boolean b;
    public String greet () {
        System.out.println("Hello!");
    }
}
```

There are four levels, in decreasing order of security:

1. **private** — Can be used only within class C.
2. **default/package level** (no keyword used) — Can be used within C and any class in the same package as C.
3. **protected** — Can be used within C, any class in C’s package, and any inheritance descendant of C in other packages.
4. **public** — Can be used by any client of C.

## Accessor Methods

- Sometimes, it is tempting to make a variable `public` and let clients change/access it directly.
- Years and years of industrial experience has shown that this is a really bad idea, so resist the temptation:
  - Clients come to depend upon the representation of the object, so you can’t change it easily without breaking code elsewhere.
  - You want clients to rely on the **abstraction** the class represents, not the particular representation (which might change?)
- Solution: Define **accessor** methods for each such variable:
  - Declare simple methods to get/set values of these variables; can combine several get/set actions into one method.
  - Ideally, you should pass out **values**, not references to ongoing objects.

### Accessor Methods: Example

- A graphical object with x and y fields.

```java
class graphicalObject {
    private int x, y;
    public void setLoc (int newX, int newY) {
        x = newX;
        y = newY;
    }
    public int getX () { return x; }
    public int getY () { return y; }
    ...}
```

- Because clients are using functions instead of raw variables, easier to fix/fudge if you change your mind about the representation.

## Java Packages

- Packages are a grouping mechanism at a higher level than classes.
  - A package is a named group of classes that “belong together”.
- Each package identifies which package it is contained in:

```java
package hockeyStats;  
public class hockeyPlayer {
    ...}
```

If you omit the `package` statement, then the class is assumed to belong to “the unnamed package”.

- For our use, most of the code you will write will go into a single, unnamed package. You will also use other packages by importing them as needed.
• If you want to use classes defined in another package, then:
  1. The class must be defined as public, and
  2. you must either import the package, or use the fully-qualified class name:

```java
class TokenReader {
    ...
    // parser for current input line
    // (StringTokenizer is from java.util)
    private StringTokenizer T;
    ...
}
```

• The Java API defines quite a few packages, e.g., java.util, java.awt.event, java.io. The java.lang package is always automatically imported for you; all others must be imported explicitly at the beginning of each class if you want to use classes in that package.

---

**Java Class-Level Visibility**

• Any class is visible in the package in which it is defined. That is, any other class in the same package may:
  • instantiate from it.
  • access its non-private variables and methods.

• To be visible to classes in different packages, a class must be defined as public:

```java
public class C {
    ...
}
```

If a class D is in a different package than that of C, then
• D may access C’s public variables and methods.
• D may access C’s protected variables and methods if D inherits (transitively) from C.

• No such thing as protected or private classes.

---

**Modularity**

• The killer problems in building BIG systems are:
  • Overall complexity of system; HUGE amounts of detail all of which MUST be managed correctly.
  • Fragility and brittleness of components and connections.
  • Inscrutability of components and connections.

• The key technique is use of reliable and enforceable ABSTRACTION.
  • Make individual components into coherent entities.
  • System components keep secrets from each other (e.g., details of internal representation; expose only “abstract buttons”)
  • Tightly limit “special knowledge” that is allowed to circulate.

---

**Information Hiding**

Key idea: “information hiding” [Parnas]

• In the traditional (pre-OO) view, there are just two levels of visibility: public and private.
  • Separate interface\(^a\) (abstraction) from implementation (grungy details) as much as possible.
  • Make interface public; interface to represent the abstract entity; what it is, not how it happens to be implemented.
  • Make implementation private; only component itself can touch inner working.

• In object-oriented systems, we have a sliding scale of visibility (private, default, protected, public).
  • Effectively, these define a hierarchy of interfaces reflecting trustedness of clients: self, same package, same package + descendants, general public.

\(^a\)Java language interfaces are a slightly different idea.
The idea is also referred to as “black box abstraction”:

Q: How much electronics knowledge do you need to use a TV remote or VCR?

Q: How much mechanical engineering do you need to be able to use a soda machine?

Think of the public interface as defining a contract between a client and a service provider:

→ Contract/interface states exactly what services a client may expect.

→ Service provider decides how to do it.

Information hiding:

- “The client should be told only as much information as (s)he needs to know to use the module and nothing more. — [Parnas]

- principle of minimum disclosure, “need-to-know”

Why hide information?

→ Greatly simplifies large programs. Cleaner, simpler interactions between system components. More system cohesion; looser coupling between components.

→ Clients don’t have to worry about hairy details. Easier to use, less to learn, visible “essence” rather than “detail”.

→ Often some performance overhead, ‘tho ...

Why hide information?

→ Contains and isolates hairy details; clients are less likely to “break” the code here or elsewhere.

→ Can later change internal details without client needing to know (e.g., more efficient algorithm or data representation ... but don’t change the interface!)

→ ideally, implementor should be able to replace implementations without the user noticing

  e.g., stack implemented as linked list vs. array

→ Managing (and minimizing) complexity of structure and interactions is key to building big systems!

Q: What should the interface to a stack class look like? What’s essential? What’s an implementation detail?

The Role of the Package

- Java is nearly unique\(^a\) in that it contains an enforceable grouping mechanism at a higher level than the class/module: the package.

  → Same idea elsewhere is sometimes called the subsystem, but isn’t enforced by language or tools.

- Even if you have no tool support for creating high-level views of interactions between classes/modules, software system architects still draw these diagrams for communication and understanding.

\(^a\)Well, technically, something is either unique or it isn’t; there are no half measures to uniqueness.
System hierarchy of a real (small) operating system — blue (atomic) boxes are modules/classes; grey (composite) boxes are subsystems.

How do you do ...

How do you design these hierarchies structures?

- Put like-things in the same box; put different things in different boxes
- Allow some “mutual understanding” between like things.
- Insist that different things communicate only by established protocols, well-designed interfaces.

Examples in practical use:

- Java packages, Java and C++ private/protected/public entities
- Netscape plug-ins
- IDLs/CORBA, Java Beans/RMI, OLE/ActiveX/COM/DCOM
- client/server system architectures