CSCE 314: Programming Languages

Dr. Dylan Shell

Classes, subclasses, subtyping

‘Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects... It is what I sometimes have called “the separation of concerns”, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of.’

— *Edsger Dijkstra*

On the role of scientific thought, August 1974

Abstract Data Types (ADTs)

- Object-oriented programming has its roots in ADTs
- ADTs
  - Encapsulate state along with a set of operations
  - Specify an interface for a data type hiding the underlying type representing the state is not directly accessible
  - Allow multiple implementations of the same ADT
- We saw examples of ADTs in Haskell, built with the help of Haskell’s `module` construct
- Many language features can serve for implementing ADTs

```c
// stack.h

struct node
{
  int data;
  struct node *next;
};

typedef struct node* STACK;

int   empty(STACK s);

STACK newstack();

int   pop(STACK* s);

void  push(STACK* s, int x);

int   top(STACK s);
```

```c
// stack.c

#include "stack.h"
#include <stdlib.h>

int empty(STACK s) { return s == 0; }

STACK newstack() { return (STACK) 0; }

int pop(STACK* s) {
  STACK tmp;
  int res = (*s)->data;
  tmp = *s;
  *s = (*s)->next;
  free(tmp);
  return res;
}

void push(STACK* s, int x) {
  STACK tmp;
  tmp = (STACK)malloc(sizeof(struct node));
  tmp->data = x;
  tmp->next = *s;
  *s = tmp;
}

int top(STACK s) { return s->data;}
```

Levels of Abstraction

- ADTs in C (using struct) do not really hide the underlying data-types
- Stronger module/package systems of, for example, CLU, ML or Ada fully hide the type (CLU and Ada were major inspirations for C++ templates)
- Parameterized ADTs via many mechanisms
  - Java generics
  - Ada packages
  - C++ templates
  - Haskell modules + parameterized data types
  - ML Functors
  - . . .

From ADTs to Classes (and Object-Oriented Programming)

- ADTs don’t give a clear answer to
  - Automatic initialization (allocating memory, opening files, initializing local variables, ...) and finalization
  - Reuse between similar ADTs
- *Classes* and *inheritance* provide one answer
- Object-oriented programming adds the metaphor of network of objects communicating via sending and receiving messages

E.g. Simula-67: big influence on Stroustrup to develop C++

Inheritance

- Inheritance in OO is based on the idea that ADTs have a lot in common
- Lends to a hierarchical structure of ADTs

  for example: *arrays* and *lists* are both *sequences*
- Inheritance enables hierarchical definition of ADTs
- Assume ADT **B** has substantially the same functionality as ADT **A**.
  - If **B** is defined to inherit from **A**, it suffices to encode the difference between their functionalities.

OO Definitions

- There are many definitions. At least:
- OOP = encapsulated state + inheritance (with dynamic binding)
- An object is an entity that
  1. has a unique identity
  2. encapsulates state
- State can be accessed in a controlled way from outside by means of *methods* that have direct access to state.
- State is also initialized and finalized in a controlled way.

Class

- Blueprint from which individual objects are created
- A unit of specification of objects in an incremental way
    - achieved by declaring inheritance from other classes and by encoding the difference to inherited classes, for example:

Bicycle {cadence, speed, gear, ...}

Mountain Bike *Road Bike* *Tandem Bike*

Note: OO languages that do not have the notion of a class exist (e.g. JavaScript)

Class Invariant

A logical condition that ensures that an object of a class is in a well-defined state

- Every public method of a class can assume the class invariant in its precondition.
- Every public method of a class must ensure that the class invariant holds when the method exits.

Class Invariant

A logical condition that ensures that an object of a class is in a well-defined state

- Every public method of a class can assume the class invariant in its precondition.
- Every public method of a class must ensure that the class invariant holds when the method exits.

E.g., class triangle { double a, b, c; . . . };

Class Invariant

A logical condition that ensures that an object of a class is in a well-defined state

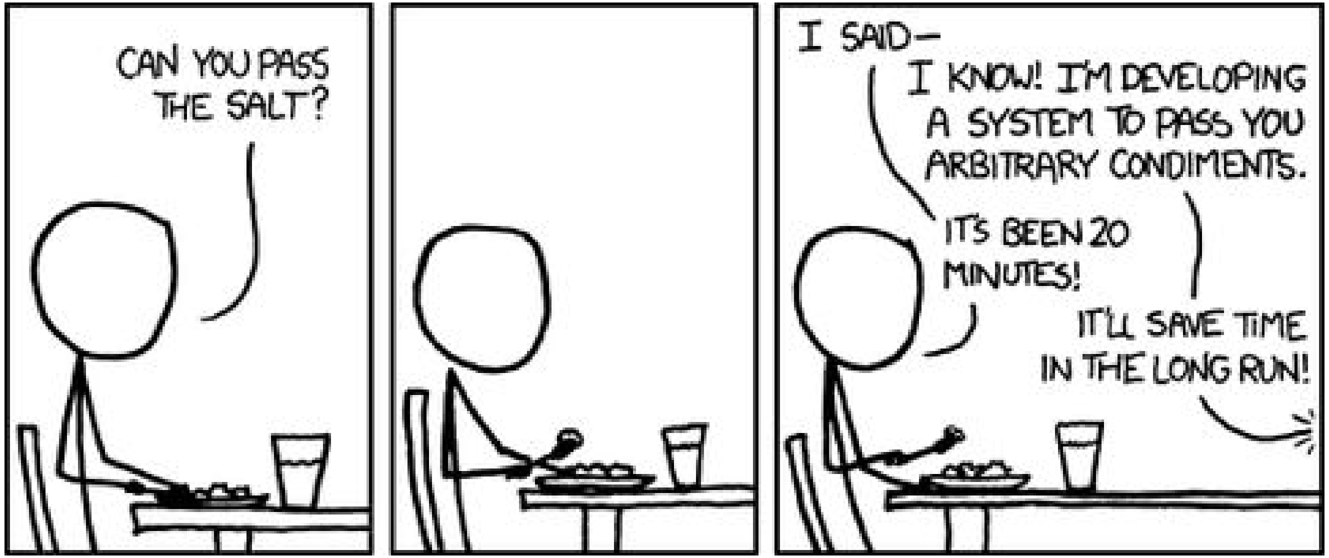
- Every public method of a class can assume the class invariant in its precondition.
- Every public method of a class must ensure that the class invariant holds when the method exits.

E.g., class triangle { double a, b, c; . . . };

Invariant: a, b, c > 0 and a+b>c and a+c>b and b+c>a

Caveat

- Object-oriented programming may have once been thought as the “Silver Bullet”
- It’s not! Many problems arise with the size of the software
- OOP can lead to networks of objects with sharing (aliasing) all over the place. Reasoning about such systems is difficult, reuse opportunities don’t realize, ...
- Researchers still have work to do, and software professionals still have new languages and new paradigms to learn


CAN YOU PASS THE SALT?
I SAID–
I KNOW! I'M DEVELOPING A SYSTEM TO PASS YOU ARBITRARY CONDIMENTS.
IT'S BEEN 20 MINUTES!
IT'LL SAVE TIME IN THE LONG RUN!

XKCD

Getting Started with Java

- Classes
- Interfaces
- Inheritance

Short History of Java

- Originally known as Oak
  - first prototype Sep 1992 by the Green Team (Sun)
  - independently of World Wide Web
  - for distributed, heterogeneous network of consumer electronic devices
  - (commercial) failure
- Officially announced on May 23, 1995
  - incorporated in Netscape Navigator

Aims

- Platform independence
  - Java Virtual Machine
- Built-in support for computer networks
- Execute code from remote sources
- Use Object Oriented Programming methodology
- and more

Hello World!

```
HelloWorld.java

      class HelloWorld
     {
       public static void main(String args[])
       { System.out.println("Hello World!"); }
     }
> javac HelloWorld.java
> java HelloWorld
```

Assumption: you know the basics of Java (or C++)

What do “public” and “static” mean above?

Hello World!

```
HelloWorld.java
      class HelloWorld
     {
       public static void main(String args[])
       { System.out.println("Hello World!"); }
     }
> javac HelloWorld.java
> java HelloWorld
```

so that anyone can invoke it

the method belongs to the class (not associated with a particular instance of the class)

Assumption: you know the basics of Java (or C++)

What do “public” and “static” mean above?

Java Basics

- How to edit, compile, and run Java programs
- Java’s fundamental data types
- Java’s control structures
- Java’s expressions
- How to declare variables, construct objects
- I/O
- importing packages
- Arrays
- Java’s scoping rules

Fundamental Data Types

Primitive data types: boolean, char, byte, short, int, long, float, double

Each with corresponding “wrapper” class:

$$\begin{tikzcd}
 & & \texttt{Object} & & & \\
\texttt{Boolean} \arrow[urr] & \texttt{Character} \arrow[ur] & & \texttt{Number} \arrow[ul] & & \texttt{Void} \arrow[ulll] \\
\texttt{Byte} \arrow[urrr] & \texttt{Short} \arrow[urr] & \texttt{Integer} \arrow[ur] & \texttt{Long} \arrow[u] & \texttt{Float} \arrow[ul] & \texttt{Double} \arrow[ull]
\end{tikzcd}$$

Example Class: Stack

```
public class Stack {
  protected class Node {
    int data;
    Node next;
    Node (int v, Node n) { data = v; next = n; }
  }
  public Stack() { stk = null; }
  public boolean empty() { return stk == null; }
  public int pop() {
    int result = stk.data;
    stk = stk.next;
    return result;
  }
  public int top () { return stk.data; }
  public void push (int i) { stk = new Node (i, stk); }
  private Node stk; // state variable, properly encapsulated
}
```

Notes on Java Specifics

- The state variables—only `stk` here—are properly encapsulated: only methods of `Stack` have access to it and its type
- Class now defines initialization (nothing special has to occur in finalization)
- No need to worry about releasing memory: Garbage collection
- Caveat: Garbage collection deals **only** with memory. All other resources (GUI handles, file handles, ...) must be managed by programmer (finalization is a hard problem)
- access protection (`public`, `private`, `protected`) per each member
- inner class
- no special member initializers syntax like that of C++

Instantiating and Invoking Class Members

```
class StackMain {
  public static void main (String args[])
  {  Stack s = new Stack();
     s.push(1);
     s.push(3);
     s.pop();
     System.out.println( Integer.toString( s.top() ) );
  }
}
```

Note: static/class methods vs. instance methods

Access Control

How programming language restricts access to members of objects or classes.

- Java: public, private, protected, and “package” (no modifier)
- C++: public, protected, private

The meaning of access control modifiers vary from one language to another

- e.g., whether attributes of another object of the same type is accessible or not

ADTs with Classes

- Classes provide encapsulation of state
- To implement ADTs with classes, we need the notion of an interface
- Mechanisms vary
  - C++, Eiffel, Java, C#: abstract classes
  - Java, C#: interfaces
  - Scala: traits
- `interface` and `trait` specify purely an interface of an ADT, abstract classes may have other uses (code reuse)
- `interface`s and `trait`s are *stateless*

Interfaces

- Interface is like a class definition, except for no method bodies or instance variables. Example:

```
public interface IStack {
  public boolean empty();
  public int pop();
  public int top ();
  public void push (int i);
}
```

- We can now plug-in many different implementations for the same interface:

```
public class Stack implements IStack { ... }
public class AnotherStack implements IStack { ... }
```

Interfaces as ADTs

- An `interface` gives an interface against which to write code that is oblivious of the implementation of the interface
- Given the following classes:

```
class Coin {
  public getValue() { . . . }
  . . .
}
```

```
class File {
  public getSize() { . . . }
  . . .
}
```

Task: Implement containers DataSet that keep track of the maximal and accumulated file sizes or values of coins

```
class DataSet {
  ...
  public add(Coin x) {
    total = total + x.getValue();
    if (count == 0 ||
        max.getValue() < x.getValue())
      max = x;
    count++;
  }
  public Coin getMax() {
    return max;
  }
  private double total;
  private Coin max;
  private int count;
}
```

```
class DataSet {
  ...
  public add(File x) {
    total = total + x.getSize();
    if (count == 0 ||
        max.getSize() < x.getSize())
      max = x;
    count++;
  }
  public File getMax() {
    return max;
  }
  private double total;
  private File max;
  private int count;
}
```

```
public interface Measurable {  double getMeasure();  }

class Coin implements Measurable {          class File implements Measurable {
  public getMeasure { return getValue(); }    public getMeasure { return getSize(); }
  public getValue() { . . . }                 public getSize() { . . . }
  . . .                                       . . .
}                                           }
```

```
public interface Measurable {  double getMeasure();  }

class Coin implements Measurable {
  public getMeasure {return getValue();}
  public getValue() { . . . }
  . . .
}
```

```
class File implements Measurable {
  public getMeasure {return getSize();}
  public getSize() { . . . }
  . . .
}
```

```
class DataSet {
  ...
  public add(Coin x) {
    total = total + x.getValue();
    if (count == 0 ||
        max.getValue() < x.getValue())
      max = x;
    count++;
  }
  public Coin getMax() {
    return max;
  }
  private double total;
  private Coin max;
  private int count;
}
```

```
class DataSet {
  ...
  public add(Measurable x) {
    total = total + x.getMeasure();
    if (count == 0 ||
        max.getMeasure() < x.getMeasure())
      max = x;
    count++;
  }
  public Measurable getMax() {
    return max;
  }
  private double total;
  private Measurable max;
  private int count;
}
```

Substitutability via Subtyping

- We can use a `Coin` or a `File` where a `Measurable` is expected because of *subtyping* and *substitutability*
- `class Coin implements Measurable` establishes that `Coin` is a subtype of `Measurable`
- Symbolically, `Coin <: Measurable`
- Substitutability: If `S <: T`, then any expression of type `S` can be used in any context that expects an expression of type `T`, and no type error will occur. As a type rule

$$\frac{\Gamma \vdash e : S \qquad S <: T}{\Gamma \vdash e : T}$$

Abstract Classes and Methods

- An abstract class is a class declared abstract
  - it may or may not include abstract methods
  - it cannot be instantiated
- An abstract method is a method declared without body
- If a class includes abstract methods, then the class must be declared abstract, example:

```
public abstract class Shape {
     // declare fields
     // declare nonabstract methods
     abstract void draw();
  }
```

Abstract Classes Compared to Interfaces

- With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default or static methods) are public
- Abstract classes can have fields that are not static and final, and public, protected, and private concrete methods
- A class can extend only one class, whether or not it is abstract, whereas it can implement any number of interfaces
- An interface can “extend” (but cannot “implement”) multiple interfaces
- An interface cannot be instantiated
- Example interfaces: `Comparable`, `Cloneable`, `Serializable`, etc.

Abstract Classes Compared to Interfaces

- With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default or static methods) are public

```
public interface  Choices   {
    public static final int YES  = 1;
    public static final int NO  = 2;
}
```

-
- it can implement any number of interfaces
- An interface can “extend” (but cannot “implement”) multiple interfaces
- An interface cannot be instantiated
- Example interfaces: `Comparable`, `Cloneable`, `Serializable`, etc.

Inheritance

- Inheritance allows a (more) economical description of related ADTs
- A subclass extends a superclass.

```
class SavingsAccount extends BankAccount
{
   // new methods
   // new instance variables
}
```

- `extends` induces a subtyping relation:

```
SavingsAccount <: BankAccount
```

- Contrast with inheriting from an interface: Here, subclass inherits *behavior* and *state* from a superclass

Inheritance Hierarchy

- OO Ideal
  - a set of related classes can be easily implemented by extending other classes via inheritance
  - everything stays nicely open-ended and extensible, in case new needs arise—just add another class
- Example: List, Stack, Queue, Dequeue, PriorityQueue
- "Inheritance hierarchies"
- The inheritance relation induced by "`extends`" in Java is rooted by `Object`
- Not all languages (e.g., C++) have a dedicated root class

Code Reuse via Inheritance

```
public class List extends Object {
  protected class Node {
    public Object data;
    public int priority;
    public Node prev, next;
    public Node (Object v, Node p) {data = v; prev p; next = null; priority = 0;}
    public Node (Object v, Node p, Node n) {
      data = v; prev = p; next = n; priority = 0;
    }
    public Node (Object v, int pr, Node p, Node n) {
      data = v; prev = p; next = n; priority = pr;
    }
  }
}
```

```
public class Stack extends List {
  private Node stk;
  public Stack () { stk = null; }
  public Object pop() {
    Object result = stk.data;
    stk = stk.next;
    return result;
  }
  public void push (Object v) { stk = new Node (v, stk);}
  . . .
}
```

Code Reuse via Inheritance (Cont.)

```
public class List extends Object { . . . }
public class Stack extends List { . . . }
public class Queue extends List {
  protected Node front = null, rear = null;
  public void enter (Object v) { . . . }
  public Object leave () { . . . }
  . . .
}
public class PriorityQueue extends Queue {
  public void enter (Object v, int pr) { . . . }
  public Object leave () { . . . }
}
public class Deque extends List {
// double-ended queue, pronounced “deck”
  public void enterFront (Object v) { . . . }
  public void enterRear (Object v) { . . . }
  public void leaveFront (Object v) { . . . }
  public void leaveRear (Object v) {  . . . }
  . . .
}
```