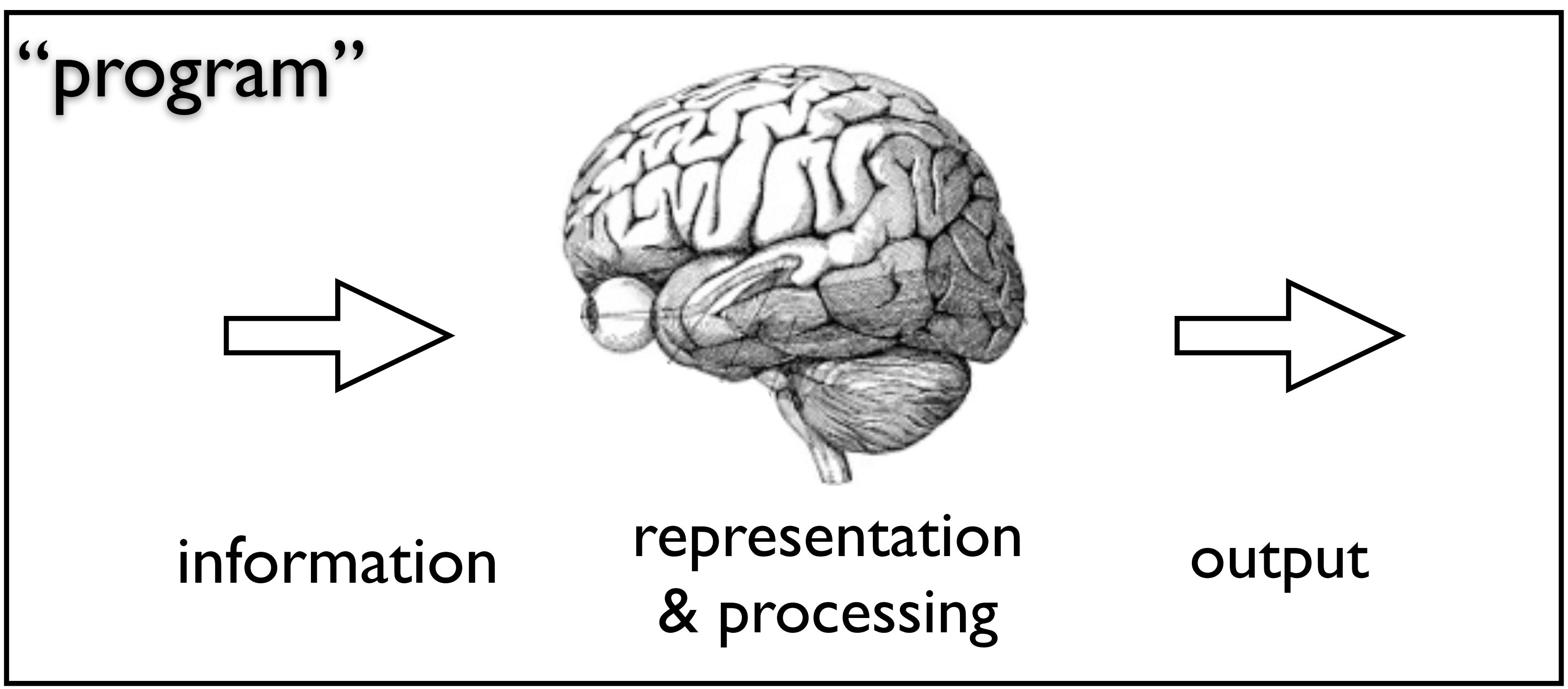
Abstract Data Types (ADTs)

EECS 233

Today

- Memory management
- Abstract Data Types (ADTs)
- Computer programming is about creating useful abstractions



First a poll

- What *have* you used to develop a java program?
- Which one do you *prefer*?

First a poll

- What *have* you used to develop a java program?
- Which one do you *prefer*?

- Examples (used before / will use):
  - DrJava (70% / 3%) - simple, few features
  - Eclipse (45% / 40%) - more complicated, powerful
  - notepad++ (25% / -)
  - textpad (15% / -)
  - NetBeans (5% / -)

- Pros/cons?

Words of wisdom

- *The design of the data structure is central to the creation of the program.*

  Kernighan and Pike

- *A fundamental principle of programming is to understand clearly what you are trying to accomplish before you set out to accomplish it.*

  *Figure out what are the operations to be performed on your data **before** you choose a representation for the data.*

  Lewis and Denenberg

Where do “data structures” lie?

Various applications (managing complex types of data), to be built by high-level application programmers

Data structures and algorithms to accomplish common tasks

Low-level memory management of modern programming languages

Memory Management in PLs

- First some low-level basic stuff...
- Programming languages provide low-level support (for basic memory management mechanisms)
  - **Static storage**: for global variables
  - **Stack storage**: for method calls (parameters, return address, and local variables)
  - **Heap storage**: for dynamic memory allocation, new()
- Knowing these concepts will help us develop better computer programs

Computer System Storage

- Data structures organize data, which is stored in computer systems
  - We need first understand computer storage management: CPU cache, memory, disk.

- Memory is a sequence of bits, an array of words (e.g., 32-bit)
  - no structure
  - hard to manage information
  - unwieldy for algorithm design

- Programming languages simplify memory management.

- Three main types of memory allocation
  - Static storage
  - Stack storage
  - Heap storage

Memory Management: Static Storage

- The region of memory allocated once at the start of the program is called *static storage*.

- In Java, it is used for *class variables* that are declared with the keyword static:

```
public static final double PI = 3.14159;

public static int numTelephones;
```

  - there is only one copy of each class variable; it is shared by all instances (i.e., all objects) of the class.

  - allocated when the class is first encountered

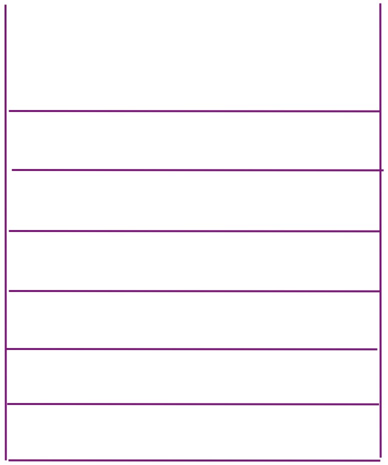
- Static storage is automatically allocated for us, and it stays fixed during the lifetime of a program.

Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack. Stack Pointer (SP) indicates the current pointer of stack memory allocation

```
public class foo {
    static int mod2(int i) {
        if (i < 2) return i;
        return mod2(i-2);
    }
    public static void main(String[] args) {
        int result = mod2(3);
        System.out.println(result);
    }
}
```



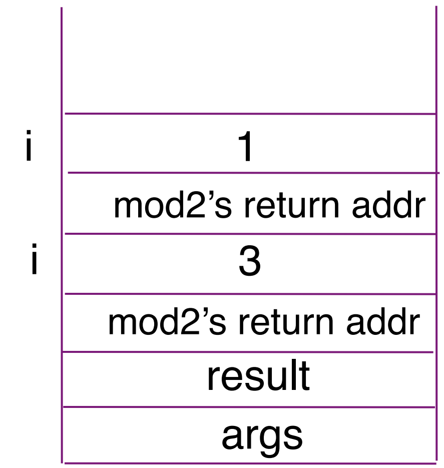
- How does the stack change as the program is executed?

Memory Management: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.

- For each method call, a new *stack frame* is added to the top of the stack. Stack Pointer (SP) indicates the current pointer of stack memory allocation

```
public class foo {
    static int mod2(int i) {
        if (i < 2) return i;
        return mod2(i-2);
    }
    public static void main(String[] args) {
        int result = mod2(3);
        System.out.println(result);
    }
}
```



- When a method completes, its stack frame is removed. The values stored there are *not* preserved.

Memory Management: Heap Storage

- The third region of memory is known as the *heap* or free store

- Memory on the heap is allocated using the new operator:

```
int[ ] values;
values = new int[3];
MyClass b = new MyClass();
```

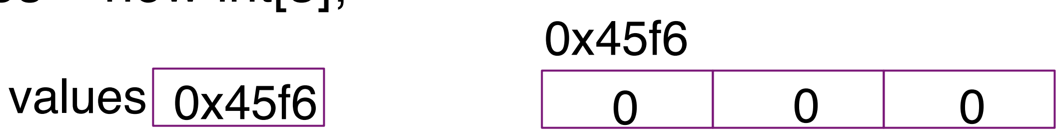
- This type of memory allocation is often referred to as *dynamic memory allocation*, because it is specified as part of the program and is determined at runtime:

```
System.out.println(“How many values? ”);
Scanner input = new Scanner(System.in);
int num_values = input.nextInt();
int[] values = new int[num_values];
```

Heap Storage: References and Pointers

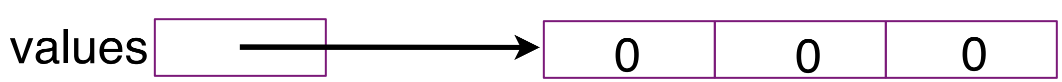
- When we allocate memory on the heap, new returns the memory address of the start of the array or object on the heap.
- This memory address is stored in the variable that represents the array/object:

  values = new int[3];



- A memory address is referred to as:
  - (a *pointer* in C++)
  - a *reference* in Java

  We will typically use an arrow to represent a reference/pointer:



Heap Storage: Deallocating Memory

- Java:
  - An object persists until there are no remaining references to it
  - Unused objects are *automatically* reclaimed by a process known as *garbage collection*,
  - This makes their memory available for other objects.

- When garbage collector fails to reclaim memory, you can eventually run out of room on the heap.
  - Memory leak!

Heap Storage: A Frequent Gotcha

- Example: what do things look like in memory when the following lines are executed?

```
int[] values = {4, 23, 3, 14};
int[] other = values;
other[2] = 30;
```

What is values[2]?

Be careful! When we copy the value of a variable that holds a reference, we get a second reference to the same object or array. We do *not* get a copy of the object/array itself.

Memory Allocation versus Data Structures

- Programming languages support various memory allocation methods

- How to organize the information is left to us
  - Programmers
    - ✓ Use existing data structures
    - ✓ Provide common libraries
    - ✓ Combine known data structures to represent complex information
    - ✓ Act as computer scientists
  - Computer Scientists (like Donald Knuth, Case grad)
    - ✓ Design new data structures
    - ✓ Design new algorithms or methods to manipulate the data structures
    - ✓ Analyze them

Abstract Data Types

- Need an interface between…
  - …"common data structures" and high-level programming
  - …different high-level objects (modular development)

- An *abstract data type* (ADT) is the model of a data structure that specifies:
  - What data (or what characteristics of objects of this type) are defined
  - What operations can be performed on the data.

- To implement an ADT, we need to design data structures (to organize the data/information) and algorithms (to describe the procedures to complete desired tasks)
  - The objective of this course!

Some examples

- **data types**: integers, reals, booleans, ...
- **operations**: add, multiply, divide, compare, ...
- each data type has its own set of associated operations
  - *strings*: concatenate, compare, display, multiply?
  - others? What might you want to do to a string?
- **abstract data types**: lists, stacks, sets, graphs
- **operations**: add, remove, contains
- *ADTs extend the language foundations*

An Example: The Bag ADT

- As the name suggests, a bag is just a container for a group of data items.
  - Analogy: a bag of candy

- Some characteristics of a bag:
  - The positions of the data items don't matter (unlike a sorted list).
    - ✓{3, 2, 10, 6} is equivalent to {2, 3, 6, 10}
  - The items do *not* need to be unique (unlike a set).
    - ✓{7, 2, 10, 7, 5} is a bag but not a set
  - Maybe has a limited size?

The Bag Operations

- Operations supported by the Bag ADT:
  - **add**(item): add item to the bag
  - **remove**(item): remove one occurrence of item (if any) from the bag
  - **contains**(item): check if item is in the bag
  - **grab**(): get an item at random, without removing it
  - **numItems**(): get the number of items in the bag
  - **toArray**(): get an array containing the current contents of the bag
  - others: isFull? weight?
- The operations are provided to the programmer.

- **Note**: we don't specify *how* the bag will be implemented.

The Bag Interface

- In Java, we can specify an ADT using an interface:

```
public interface Bag {
    boolean add(int item);
    boolean remove(int item);
    boolean contains(int item);
    …
}
```

- An interface specifies a set of methods.
  - it includes only the method headers
  - it *cannot* include the actual method definitions (i.e. bodies)

- To implement the ADT, we define a class that implements the interface:

```
public class MyBag implements Bag {
      …
}
```

  - Need data structures and algorithms (procedures/functions)

Other examples

- set
  - What are the fields and methods?
  - How would you implement it?
- shape
  - Methods? Fields? Implementation?
- choice of methods depend on what you want to ***do*** with the objects

The Bag Implementation: Design Choices

- Bags can contain integers only.
  - We will consider a more general implementation later.
- We will use an array to store the items.
  - The array is *not* mandated by the ADT.
  - Other design choices are possible (e.g., linked list).

Implementation of IntBag in Java

```
public class IntBag { // bag of ints for now
// instance variables (also known as fields, members, attributes, properties)
    private int[] items;  // items is a reference
    private int numItems;
    …
    // methods (also known as functions)
    public boolean add(int item) {
        if (numItems == items.length)
            return false; // no more room!
        else {
            items[numItems] = item;
            numItems++;
            return true;
        }
    }
    …
}
```

Accessing Private Fields

```
public class IntBag { // bag of ints for now
  // instance variables (also known as fields,
     members, attributes, properties)

  private int[] items; //items is a reference
  private int numItems;
  …
  // methods (also known as functions)
  public boolean add(int item) {
     if (numItems == items.length)
           return false; // no more room!
        else {
           items[numItems] = item;
           numItems++;
           return true;
        }
     }
     …
}
```

- Private fields are for the internal use by the implementation
  - Not exposed to ADT users.
  - Collectively form the data structures.

- A method can access a private field of its own object

```
public boolean addAll(IntBag other) {
     for (int i = 0; i < other.numItems; i++)
          add(other.items[i]);
}
```

- … or *other* objects *from the same class* that are passed to the method as parameters.

Encapsulation

- *Encapsulation* is one of the key principles of object-oriented programming.
  - Also known as *information hiding*
    - It refers to "hiding" the implementation of a class from users of the class.
  - Prevents *direct* access to the internals of an object
  - Provides controlled, limited, *indirect* access through a set of methods

- Black Box Analogy
  - We treat many objects in our lives as "black boxes."
  - We know *what* operations they can perform, but we don't know *how* they perform those operations
  - We can't see inside the box, so we interact with them using a well-defined *interface*

- Power of abstraction!

Encapsulation (cont.)

- Java uses *private* instance variables (and occasionally private helper methods) to hide the implementation of a class.
  - these private members can only be accessed inside methods that are part of the same class

```
class MyClass {
    …
    void myMethod() {
        IntBag b = new IntBag();
        b.items[0] = 17; // not allowed!
    …
```

- Users are limited to the *public* methods of the class, as well as any public variables (usually limited to constants – why?).
  - public members can be accessed inside methods of *any* class

Benefits of Encapsulation

- It prevents inappropriate changes to the state of an object:

```
class MyClass {
   …
   void myMethod() {
      IntBag *b = new IntBag();
      b->addItem(7);
      b->addItem(22);
      b->numItems = 0; // not allowed
   …
```

- Can you think of another benefit?

- Please make sure to use proper encapsulation in the classes that you write for this course!

Summary

- Programming languages support data structures/ algorithm design
  - From low-level memory management to templates/ generics
- ADTs are interfaces; data structures/algorithms are needed for the implementation
  - Good ADT implementations require good data structures and efficient algorithms
  - Implementing good data structures and algorithms as ADTs facilitates their usage by others.
- Next week
  - Some basic mathematics, recursion, and algorithm complexity (e.g. big O)