Welcome to Computer Science E-119!

• We will study fundamental data structures.
  • ways of imposing order on a collection of information
  • sequences: lists, stacks, and queues
  • trees
  • hash tables
  • graphs

• We will also:
  • study algorithms related to these data structures
  • learn how to compare data structures & algorithms

• Goals:
  • learn to think more intelligently about programming problems
  • acquire a set of useful tools and techniques
Sample Problem I: Finding Shortest Paths

- Given a set of routes between pairs of cities, determine the shortest path from city A to city B.

Sample Problem II: A Data "Dictionary"

- Given a large collection of data, how can we arrange it so that we can efficiently:
  - add a new item
  - search for an existing item

- Some data structures provide better performance than others for this application.

- More generally, we’ll learn how to characterize the efficiency of different data structures and their associated algorithms.
Prerequisites

- A good working knowledge of Java
  - comfortable with object-oriented programming concepts
  - some prior exposure to recursion and linked lists would be helpful
  - if your skills are weak or rusty, you may want to consider first taking CSCI E-50b/S-111a

- Familiarity with precalculus mathematics (Math E-10)
  - several topics will involve mathematical reasoning
  - mostly simple algebra, but need to understand the basics of logarithms (we’ll review this)
  - will do some simple proofs

Requirements

- Lectures and weekly sections
  - sections: start next week; times and locations TBA
  - also available by streaming and recorded video

- Five problem sets
  - plan on 10-20 hours per week!
  - code in Java; must compile on nice.fas.harvard.edu
  - must be your own work
  - grad-credit students will do extra problems

- Open-book midterm exam

- Open-book final exam

- Programming project
  - for grad credit only
Additional Administrivia

• Instructor: Dave Sullivan
  • lecturer on computer science, Boston University
  • CSCI S-111 since 2000; CSCI E-119 since 2005

• TAs: Cody Doucette, Ryan Meltzer, Lily Wong

• Office hours and contact info. will be available on the course Web site:
  http://www.courses.fas.harvard.edu/~cscie119

• For questions on content, homework, etc., send e-mail to:
  cscie119@fas.harvard.edu

Review: What is an Object?

• An *object* groups together:
  • one or more data values (the object's *fields* – also known as *instance variables*)
  • a set of operations that the object can perform (the object's *methods*)

• In Java, we use a *class* to define a new type of object.
  • serves as a "blueprint" for objects of that type
  • simple example:

```java
public class Rectangle {
  // fields
  private int width;
  private int height;
  // methods
  public int area() {
    return width * height;
  }
  ...
```
Class vs. Object

• The Rectangle class is a blueprint:

```java
public class Rectangle {
    // fields
    private int width;
    private int height;
    // methods
    ...
}
```

• Rectangle objects are built according to that blueprint:

<table>
<thead>
<tr>
<th>width</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>55</td>
<td>72</td>
</tr>
<tr>
<td>40</td>
<td>13</td>
</tr>
</tbody>
</table>

(You can also think of the methods as being inside the object, but we won't show them in our diagrams.)

Creating and Using an Object

• We create an object by using the `new` operator and a special method known as a constructor:

```java
Rectangle r1 = new Rectangle(10, 30);
```

• Once an object is created, we can call one of its methods by using `dot notation`:

```java
int a1 = r1.area();
```

• The object on which the method is invoked is known as the called object or the current object.
Two Types of Methods

• Methods that belong to an object are referred to as instance methods or non-static methods.

  • they are invoked on an object
    
    \[ \text{int } a1 = \text{r1.area}(); \]
  
  • they have access to the fields of the called object

• Static methods do not belong to an object – they belong to the class as a whole.

  • they have the keyword static in their header:
    
    \[
    \text{public static int max(int num1, int num2) } \{
    \]
    
    \[
    \text{...}
    \]

  • they do not have access to the fields of the class

  • outside the class, they are invoked using the class name:
    
    \[ \text{int result } = \text{Math.max}(5, 10); \]

Abstract Data Types

• An abstract data type (ADT) is a model of a data structure that specifies:

  • the characteristics of the collection of data
  
  • the operations that can be performed on the collection

• It’s abstract because it doesn’t specify how the ADT will be implemented.

• A given ADT can have multiple implementations.
A Simple ADT: A Bag

- A bag is just a container for a group of data items.
  - analogy: a bag of candy
- The positions of the data items don’t matter (unlike a 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\} isn’t a set, but it is a bag

A Simple ADT: A Bag (cont.)

- The operations supported by our Bag ADT:
  - `add(item)`: add it to the Bag
  - `remove(item)`: remove one occurrence of it from the Bag
  - `contains(item)`: check if it is in the Bag
  - `numItems()`: get the number of items in the Bag
  - `grab()`: get an item at random, without removing it
    - reflects the fact that the items don’t have a position
      (and thus we can’t say "get the 5th item in the Bag")
  - `toArray()`: get an array containing the current contents of the bag

- Note that we don’t specify how the bag will be implemented.
Specifying an ADT Using an Interface

• In Java, we can use an interface to specify an ADT:

```java
public interface Bag {
    boolean add(Object item);
    boolean remove(Object item);
    boolean contains(Object item);
    int numItems();
    Object grab();
    Object[] toArray();
}
```

(see ~cscie119/examples/bag/Bag.java)

• An interface specifies a set of methods.
  • includes only the method headers
  • cannot include the actual method definitions

Implementing an ADT Using a Class

• To implement an ADT, we define a class:

```java
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        ...
    }
}
```

(see ~cscie119/examples/bag/ArrayBag.java)

• When a class header includes an implements clause, the class must define all of the methods in the interface.
Encapsulation

• Our implementation provides proper *encapsulation*.
  • a key principle of object-oriented programming
  • also known as *information hiding*

• We prevent direct access to the internals of an object by making its fields *private*.

  ```java
  public class ArrayBag implements Bag {
      private Object[] items;
      private int numItems;
      ...
  }
  ```

• We provide limited *indirect* access through methods that are labeled *public*.

  ```java
  public boolean add(Object item) {
      ...
  }
  ```

All Interface Methods Are Public

• Methods specified in an interface *must be public*, so we don't need to use the keyword `public` in the interface definition.

• For example:

  ```java
  public interface Bag {
      boolean add(Object item);
      boolean remove(Object item);
      boolean contains(Object item);
      int numItems();
      Object grab();
      Object[] toArray();
  }
  ```

• However, when we actually implement one of these methods in a class, we *do* need to explicitly use the keyword `public`:

  ```java
  public class ArrayBag implements Bag {
      ...
      public boolean add(Object item) {
          ...
      }
  }
  ```
Inheritance

• We can define a class that explicitly extends another class:

```java
public class Animal {
    private String name;
    ...
    public String getName() {
        return name;
    }
    ...
}
public class Dog extends Animal {
    ...
}
```

• We say that `Dog` is a subclass of `Animal`, and `Animal` is a superclass of `Dog`.

• A class inherits the instance variables and methods of the class that it extends.

The Object Class

• If a class does not explicitly extend another class, it implicitly extends Java’s `Object` class.

• The `Object` class includes methods that all classes must possess. For example:
  • `toString()`: returns a string representation of the object
  • `equals()`: is this object equal to another object?

• The process of extending classes forms a hierarchy of classes, with the `Object` class at the top of the hierarchy:
Polymorphism

• An object can be used wherever an object of one of its superclasses is called for.

• For example:
  ```java
  Animal a = new Dog();
  Animal[] zoo = new Animal[100];
  zoo[0] = new Ant();
  zoo[1] = new Cat();
  ...
  ```

• The name for this capability is *polymorphism*.
  • from the Greek for "many forms"
  • the same code can be used with objects of different types

Storing Items in an *ArrayBag*

• We store the items in an array of type *Object*.
  ```java
  public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    ...
  }
  ```

• This allows us to store *any* type of object in the *items* array, thanks to the power of polymorphism:
  ```java
  ArrayBag bag = new ArrayBag();
  bag.add("hello");
  bag.add(new Double(3.1416));
  ```
Another Example of Polymorphism

- An interface name can be used as the type of a variable.
  ```java
  Bag b;
  ```

- Variables that have an interface type can hold references to objects of any class that implements the interface.
  ```java
  Bag b = new ArrayBag();
  ```

- Using a variable that has the interface as its type allows us to write code that works with any implementation of an ADT.
  ```java
  public void processBag(Bag b) {
    for (int i = 0; i < b.numItems(); i++) {
      ...
    }
  }
  ```
  - the param can be an instance of any Bag implementation
  - we must use method calls to access the object's internals, because we can't know for certain what the field names are

Memory Management: Looking Under the Hood

- In order to understand the implementation of the data structures we'll cover in this course, you'll need to have a good understanding of how memory is managed.

- There are three main types of memory allocation in Java.

- They correspond to three different regions of memory.
Memory Management, Type I: Static Storage

- Static storage is used in Java for class variables, which are declared using the keyword static:
  ```java
  public static final PI = 3.1495;
  public static int numComparisons;
  ```
- There is only one copy of each class variable; it is shared by all instances (i.e., all objects) of the class.
- The Java runtime system allocates memory for class variables when the class is first encountered.
  - this memory stays fixed for the duration of the program

Memory Management, Type II: 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.

```
public class Foo {
  static void x(int i) {
    int j = i - 2;
    if (i >= 6) return;
    x(i + j);
  }
  public static void main(String[] args) {
    x(5);
  }
}
```

- When a method completes, its stack frame is removed. The values stored there are not preserved.
Stack Storage (cont.)

- Memory allocation on the stack is very efficient, because there are only two simple operations:
  - add a stack frame to the top of the stack
  - remove a stack frame from the top of the stack

- Limitations of stack storage:
  It can't be used if
  - the amount of memory needed isn't known in advance
  - we need the memory to persist after the method completes

- Because of these limitations, Java never stores arrays or objects on the stack.

Memory Management, Type III: Heap Storage

- Arrays and objects in Java are stored in a region of memory known as the heap.

- Memory on the heap is allocated using the `new` operator:
  ```java
  int[] values = new int[3];
  ArrayBag b = new ArrayBag();
  ```

- `new` returns the memory address of the start of the array or object on the heap.

- This memory address – which is referred to as a reference in Java – is stored in the variable that represents the array/object:

- We will often use an arrow to represent a reference:
Heap Storage (cont.)

- In Java, an object or array persists until there are no remaining references to it.

- You can explicitly drop a reference by setting the variable equal to `null`. For example:
  ```java
  int[] values = {5, 23, 61, 10};
  System.out.println(mean(values, 4));
  values = null;
  ```

- Unused objects/arrays are automatically reclaimed by a process known as garbage collection.
  - makes their memory available for other objects or arrays

Constructors for the `ArrayBag` Class

```java
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;

    public ArrayBag() {
        items = new Object[DEFAULT_MAX_SIZE];
        numItems = 0;
    }

    public ArrayBag(int maxSize) {
        if (maxSize <= 0)
            throw new IllegalArgumentException("maxSize must be > 0");
        items = new Object[maxSize];
        numItems = 0;
    }

    ...
}
```

- If the user inputs an invalid value for `maxSize`, we throw an exception.
Example: Creating Two `ArrayBag` Objects

```java
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

• After the objects have been created, here's what we have:

```
stack
  b2
  b1
  args

heap
  items
  numItems
  0

null null null null
```

Copying References

• A variable that represents an array or object is known as a `reference variable`.

• Assigning the value of one reference variable to another reference variable copies the reference to the array or object. It does not copy the array or object itself.

```java
int[] values = {5, 23, 61, 10};
ext[] other = values;
```

• Given the lines above, what will the lines below output?

```java
other[2] = 17;
System.out.println(values[2] + " * " + other[2]);
```
Passing an Object/Array to a Method

- When a method is passed an object or array as a parameter, the method gets a copy of the reference to the object or array, not a copy of the object or array itself.

- Thus, any changes that the method makes to the object/array will still be there when the method returns.

- Consider the following:

  ```java
  public static void main(String[] args) {
      int[] a = {1, 2, 3};
      triple(a);
      System.out.println(Arrays.toString(a));
  }
  
  public static void triple(int[] n) {
      for (int i = 0; i < n.length; i++) {
          n[i] = n[i] * 3;
      }
  }
  ```

Passing an Object/Array to a Method (cont.)

**before method call**

- `main`:
  - `a`:
    - `1 2 3`

**during method call**

- `triple`:
  - `n`:
    - `main`:
      - `a`:
        - `1 2 3`  

**after method call**

- `main`:
  - `a`:
    - `3 6 9`
A Method for Adding an Item to a Bag

```java
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        if (item == null)
            throw new IllegalArgumentException();
        if (numItems == items.length)
            return false;  // no more room!
        else {
            items[numItems] = item;
            numItems++;
            return true;
        }
    }
    ...
}
```

- `add()` is an instance method (a.k.a. a non-static method), so it has access to the fields of the current object.

Example: Adding an Item

```java
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
}
```

```
stack
  b
  message
  args

heap
  items
  numItems

null  null  null  null
  "hello, world"
```

Example: Adding an Item (cont.)

```java
public boolean add(Object item) {
    if (numItems + 1 > items.length) {
        return false;
    } else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```

- The method modifies the `items` array and `numItems`.
- note that the array holds a copy of the reference to the item, not a copy of the item itself.
Example: Adding an Item (cont.)

```java
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
}
```

```
public boolean add(Object item) {
    numItems++;
    return true;
}
```

- After the method call returns, `add`'s stack frame is removed from the stack.

Using the Implicit Parameter

```java
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;

    public boolean add(Object item) {
        if (item == null)
            throw new IllegalArgumentException();
        if (this.numItems == this.items.length)
            return false;  // no more room!
        else {
            this.items[this.numItems] = item;
            this.numItems++;
            return true;
        }
    }
}
```

- We can use `this` to emphasize the fact that we're accessing fields in the current object.
Determining if a Bag Contains an Item

- Let’s write the `ArrayBag contains()` method together.
- Should return `true` if an object equal to `item` is found, and `false` otherwise.

```java
______________ contains(______________ item) {
    
}
```

An Incorrect `contains()` Method

```java
public boolean contains(Object item) {
    for (int i = 0; i < numItems; i++) {
        if (items[i] != null && items[i].equals(item))
            return true;
        else
            return false;
    }
    return false;
}
```

- Why won't this version of the method work in all cases?

- When would it work?
A Method That Takes a Bag as a Parameter

```java
public boolean containsAll(Bag otherBag) {
    if (otherBag == null || otherBag.numItems() == 0)
        return false;
    Object[] otherItems = otherBag.toArray();
    for (int i = 0; i < otherItems.length; i++) {
        if (!contains(otherItems[i]))
            return false;
    }
    return true;
}
```

- We use `Bag` instead of `ArrayBag` as the type of the parameter.
  - allows this method to be part of the `Bag` interface
  - allows us to pass in `any` object that implements `Bag`
- Because the parameter may not be an `ArrayBag`, we can't assume it has `items` and `numItems` fields.
  - instead, we use `toArray()` and `numItems()`

A Need for Casting

- Let's say that we want to store a collection of `String` objects in an `ArrayBag`.
  
  - `String` is a subclass of `Object`, so we can store `String` objects in the bag without doing anything special:
    
    ```java
    ArrayBag stringBag = new ArrayBag();
    stringBag.add("hello");
    stringBag.add("world");
    ```
  - `Object` isn't a subclass of `String`, so this will not work:
    
    ```java
    String str = stringBag.grab(); // compiler error
    ```
  - Instead, we need to use casting:
    
    ```java
    String str = (String)stringBag.grab();
    ```
Extra: Thinking About a Method's Efficiency

• For a bag with 1000 items, how many items will `contains()` look at:
  • in the best case?
  • in the worst case?
  • in the average case?

• Could we make it more efficient?

• If so, what changes would be needed to do so, and what would be the impact of those changes?

Extra: Understanding Memory Management

• Our Bag ADT has a method `toArray()`, which returns an array containing the current contents of the bag
  • allows users of the ADT to iterate over the items

• When implementing `toArray()` in our `ArrayBag` class, can we just return a reference to the `items` array? Why or why not?