Abstract Data Types

Computer Science S-111
Harvard University
David G. Sullivan, Ph.D.

Congrats on completing the first half!

- In the second half, 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.

```
BOSTON
   84 | PORTLAND
   63 | CONCORD
134 | ALBANY
   49 | PROVIDENCE
    42 | WORCESTER
    49 | BOSTON
    185 | NEW YORK
```

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.
Example of Comparing Algorithms

• Consider the problem of finding a phone number in a phonebook.
• Let’s informally compare the time efficiency of two algorithms for this problem.

Algorithm 1 for Finding a Phone Number

```plaintext
findNumber(person) {
    for (p = number of first page; p <= number of the last page; p++) {
        if person is found on page p {
            return the person's phone number
        }
    }
    return NOT_FOUND
}
```

• If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case?
• What if there were 1,000,000 pages?
Algorithm 2 for Finding a Phone Number

```java
findNumber(person) {
    min = the number of the first page
    max = the number of the last page
    while (min <= max) {
        mid = (min + max) / 2       // page number of the middle page
        if person is found on page mid {
            return the person's number
        } else if the person’s name comes earlier in the book {
            max = mid – 1
        } else {
            min = mid + 1
        }
    }
    return NOT_FOUND
}
```

- If there were 1,000 pages in the phonebook, how many pages would this look at in the worst case?
- What if there were 1,000,000 pages?

Searching a Collection of Data

- The phonebook problem is one example of a common task: searching for an item in a collection of data.
  - another example: searching for a record in a database
- Algorithm 1 is known as **sequential search**.
  - also called **linear search**
- Algorithm 2 is known as **binary search**.
  - only works if the items in the data collection are sorted
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:
  ```plaintext
  width 10
  height 12
  width 55
  height 72
  width 40
  height 13
  ```

(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
    ```java
    int a1 = 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:
    ```java
    public static int max(int num1, int num2) {
    ...
    ```
  - they do not have access to the fields of the class
  - outside the class, they are invoked using the class name:
    ```java
    int result = 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 `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
  - `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();
  }
  ``

- 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) {
      ...
    }
  }
  ```

- 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.

- 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;
    }
}
```

```java
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:

```
Object
   ArrayBag
   String
   Animal
     Ant
     Cat
     Dog
```

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();
  ```
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:
  
  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:
  ```java
  values 0x23a
  ```
- We will often use an arrow to represent a reference:
  ```java
  values
  ```

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;
    }
    ...
}
```

Note: In the remainder of the lecture notes, I will **not** use the implicit parameter (`this`) unless it's necessary to do so.

- 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:
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};
int[] other = values;
```

![Diagram showing the copying of references]

• 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
```

```
 triple
   n
   main
   a
   3 6 9
```

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);
}
```

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

- add's stack frame includes:
  - `item`, which stores a copy of the reference passed as a param.
  - `this`, which stores a reference to the called/current object
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
}

• 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 boolean add(Object item) {
    ... else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```

- After the method call returns, `add`'s stack frame is removed from the stack.
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].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?
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:
  ArrayBag stringBag = new ArrayBag();
  stringBag.add("hello");
  stringBag.add("world");

• Object isn't a subclass of String, so this will not work:
  String str = stringBag.grab(); // compiler error

• Instead, we need to use casting:
  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?