Bags

Chapter 1

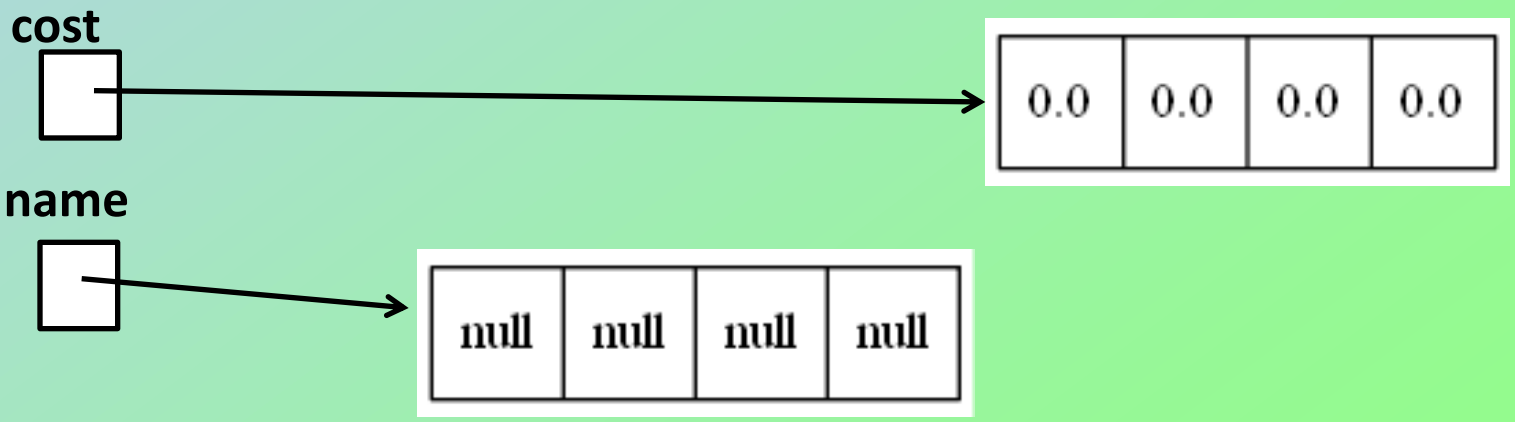
A little more about Lab 1…

- to take us into today's topics

GroceryCheckout Arrays

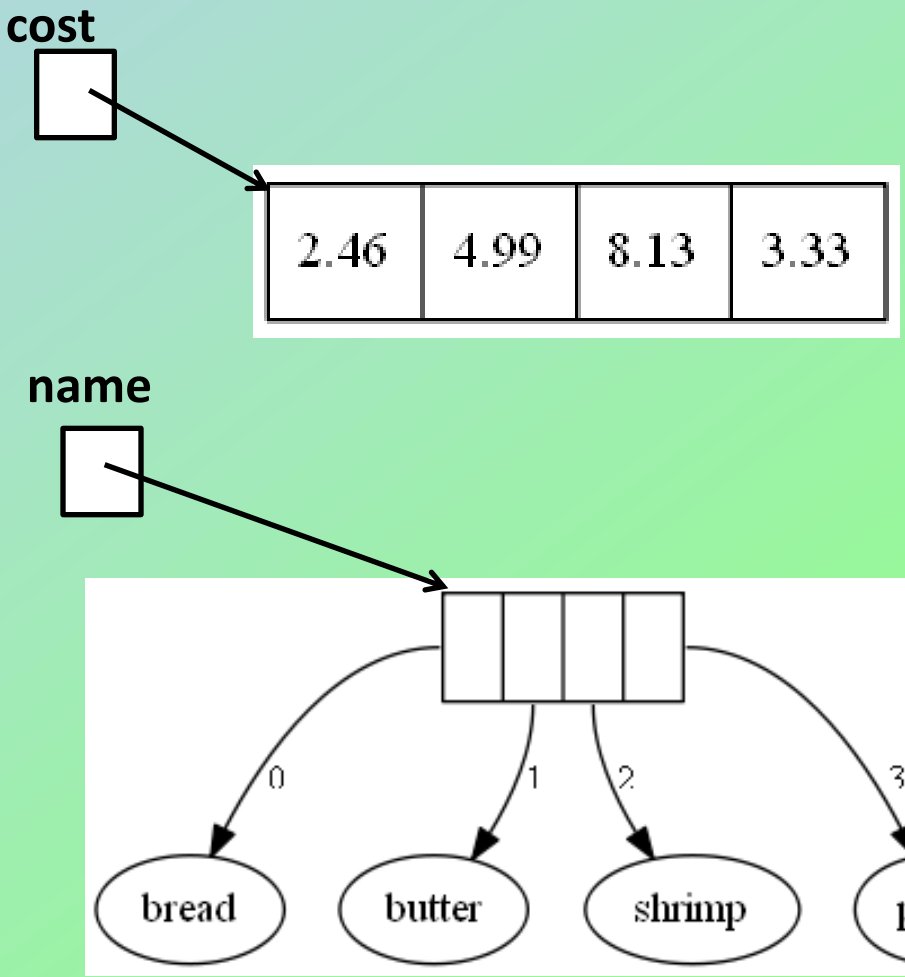
```
System.out.println("How many items? ");
numItems = keyboard.nextInt(); // enter 4

double [ ] cost = new double[numItems];
String [ ] name = new String[numItems];
```



After entering data

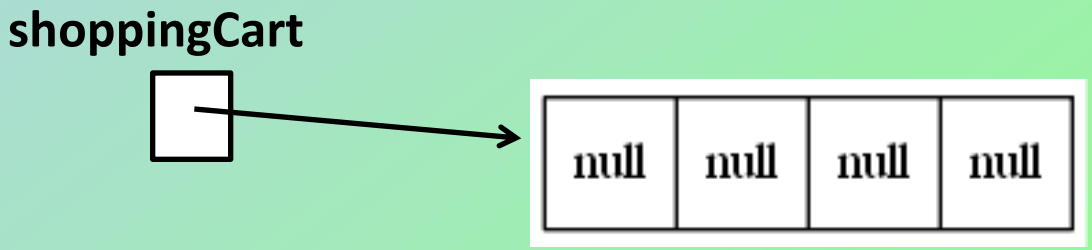
Enter the name and price:
bread 2.46
Enter the name and price:
butter 4.99
Enter the name and price:
shrimp 8.13
Enter the name and price:
peas 3.33



GroceryCheckout2 – Only 1 Array

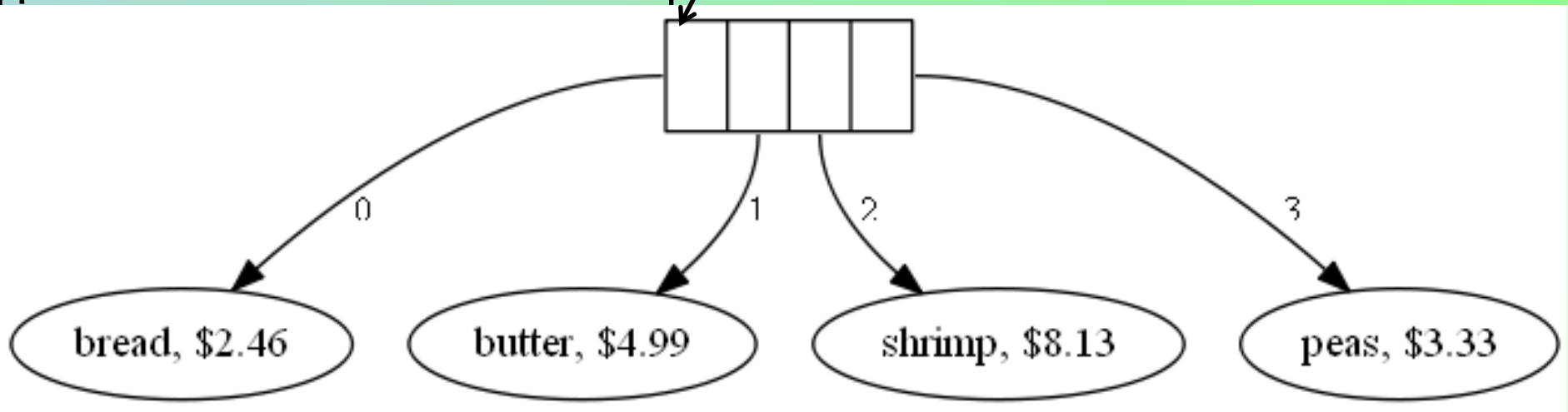
```
System.out.println("How many items? ");
numItems = keyboard.nextInt(); // enter 4

Item [ ] shoppingCart = new Item[numItems];
```



After entering data

```
Enter the name and price:
bread 2.46
Enter the name and price:
butter 4.99
Enter the name and price:
shrimp 8.13
Enter the name and price:
peas 3.33
```



Reading Quiz

1. Which of the following is not a characteristic of a Bag object?
   a. A finite collection of objects
   b. Items arranged in a ring
   c. Items in no particular order
   d. May contain duplicate items

Contents

- The Bag
  - A Bag's Behaviors
- Specifying a Bag
  - UML Diagram
- Using the ADT Bag
- Using an ADT Is Like Using a Vending Machine

Objectives

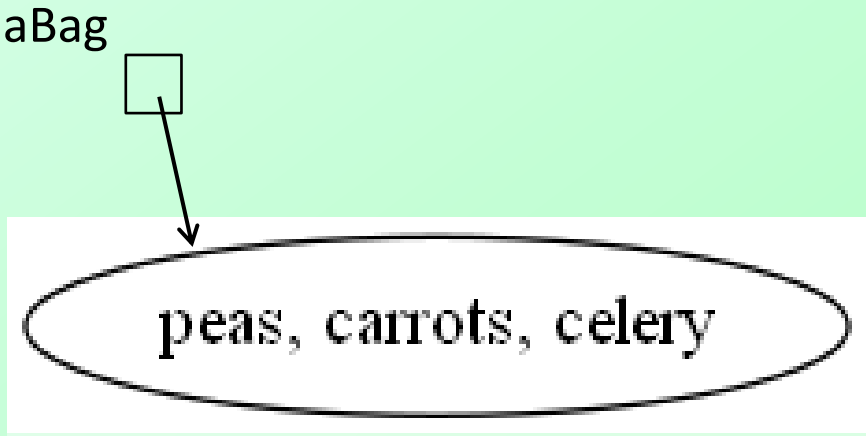
- Describe the concept of abstract data type (ADT)
- Describe ADT bag
- Use ADT bag in Java program

Definition: Bag

- A finite collection of objects
- In no particular order
- May contain duplicate items

What's in the Bag?

```
Bag<String> aBag = new Bag<String>();
```




```
aBag.add("peas");
aBag.add("carrots");
aBag.add("tofu");
aBag.add("celery");
aBag.remove("tofu");
```

Behaviors

- Determine how many objects in bag
  - Full?
  - Empty?
- Add, remove objects
- Count duplicates
- Test for specific object
- View all objects

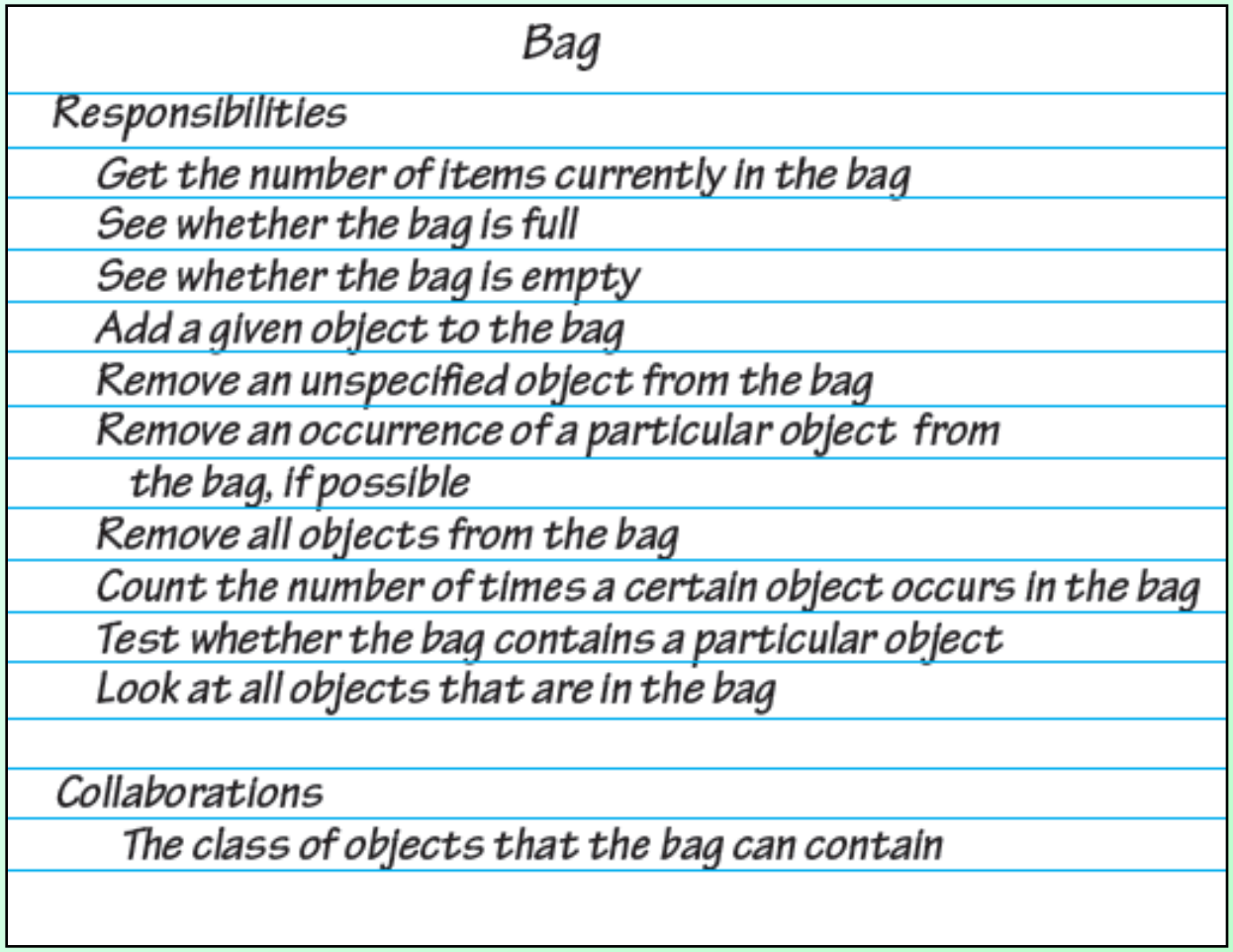


Figure 1-1 A CRC card for a class **Bag**

[Class Responsibilities and Collaborations]

Specifying a Bag

- Describe data
- Specify methods for bag's behaviors
  - Name methods
  - Choose parameters
  - Decide return types
  - Write comments

Design Decisions

- What should the method **`add`** do when it cannot add a new entry?
  - Nothing?
  - Leave bag unchanged, signal client of condition?

Design Decisions

- What should happen when an unusual condition occurs?
  - Assume invalid never happens?
  - Ignore invalid event?
  - Guess at client's intention?
  - Return flag value?
  - Return boolean value – success/failure?
  - Throw exception?

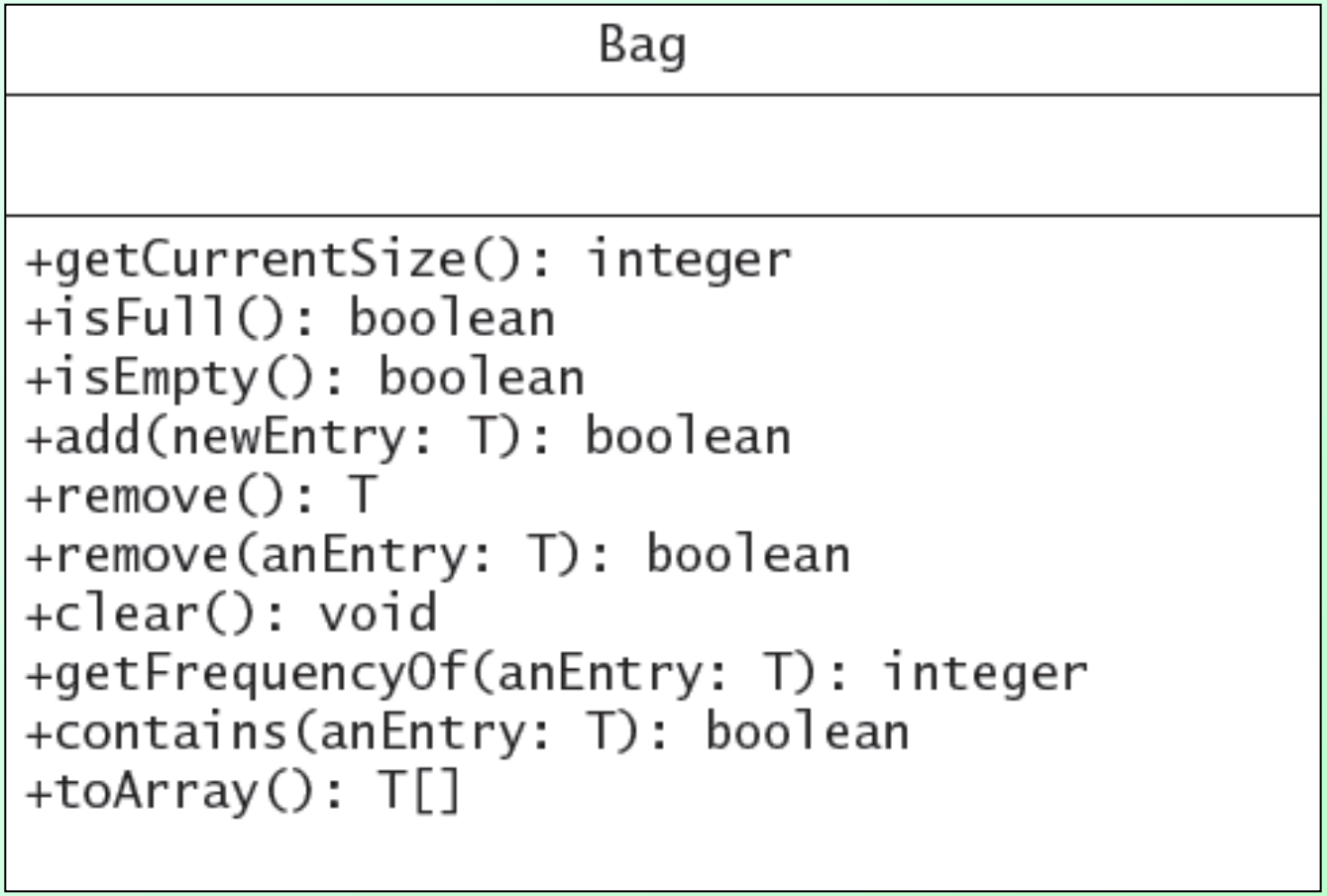


Figure 1-2 UML notation for the class **`Bag`**

Javadoc for Bag class

Class Bag<T>

A Bag is an unordered collection which may have duplicate elements.

Constructor Summary

Bag<T>()	Creates an empty bag capable of holding **25** objects of class T
Bag<T>(int capacity)	Creates an empty bag capable of holding `capacity` objects of class T

Method Summary

boolean	**add**(T newEntry) Returns true if and only if `newEntry` is successfully added to this bag
void	**clear**() Removes all entries from this bag
boolean	**contains**(T anEntry) Returns true if and only if `anEntry` is in this bag

Javadoc for Bag class (2)

int	**getCurrentSize**() Returns the number of entries in this bag
int	**getFrequencyOf**(T anEntry) Returns the number of times anEntry appears in this bag
boolean	**isEmpty**() Returns true if and only if this bag is empty
boolean	**isFull**() Returns true if and only if this bag is full
T	**remove**() Removes any non-specified entry from this bag and returns a reference to it Returns null if this bag was empty to begin with
boolean	**remove**(T anEntry) Returns true if and only if anEntry was removed from this bag
Object	**toArray**() Returns contents of this Bag as an array of Object. Entries may have to be cast back into their original type
String	**toString**() Returns a String listing of all the entries in this bag

Using ADT **Bag**

- When you need a Bag to store many instances of something in your program
  - Use the generic <type> designation

- Need a bag of words?
  - Bag<String> words = new Bag<String>();

- How about a bag of purchased items?
  - Bag<Item> shoppingCart = new Bag<Item>();

Using ADT **Bag (2)**

- Adding an item to the shopping cart:
  - shoppingCart.add(new Item("bread",2.46));

- Since the add method returns true/false you could do the following:

```
if (shoppingCart.add(new Item("bread",2.46)))
    …println("bread was added to cart");
```

- Remove and save an item from the bag
  - Item item = shoppingCart.remove( );

Using ADT **`Bag (3)`**

- Check how many items in shoppingCart:

```
…println("your cart has " +
                    shoppingCart.getCurrentSize( ) );
```

- Check if the cart contains "peas"

```
System.out.println("got peas? " +
   shoppingCart.contains(new Item("peas", 3.33));
```

How contains Method works

- The contains method traverses the bag's internal array (using a for loop)
  - returns true if there is an object in the bag's internal array that matches the target
    - "peas" in the previous example.
- Q: How do we know if there's a match
  - A: The class of objects we are storing in the bag must have a properly written equals method
  - For shoppingCart, that would be the Item class

The (wrong) equals method

- In CSIS10A, you might have written an equals method for the Item class that looks something like this (in red):

```
public class Item
{
    // instance variables
    private String name;
    private double price;

    public boolean equals( Item that ) {
          return this.name.equals(that.name) &&
                 this.price == that.price ;
    }
}
```

Unfortunately, this won't work in a Bag's contains method (read on…)

The (correct) equals method

- For CSIS10B, we have to refine our equals method definition slightly to allow for a parameter of type Object.

```
public class Item
{
    // instance variables
    private String name;
    private double price;

    public boolean equals( Object other ) {
          if ( other instanceof Item ){
              Item that = (Item) other;
              return this.name.equals(that.name) &&
                     this.price == that.price ;
          }
          else
              return false;
    }
```

A fuller explanation follows

Checkpoint 1

0) Write the Java signature for the add method based on the previous UML diagram

Question 1 Suppose `aBag` represents an empty bag that has a finite capacity. Write some pseudocode statements to add user-supplied strings to the bag until it becomes full.

Question 3 Is it legal to have two versions of `remove`, one that has no parameter and one that has a parameter, in the same class? Explain.

Question 4 Given the full bag `aBag` that you created in Question 1, write some pseudocode statements that remove and display all of the strings in the bag.

Checkpoint 1

0) Write the Java signature for the add method based on the previous UML diagram **boolean add(T newEntry);**

Question 1 Suppose `aBag` represents an empty bag that has a finite capacity. Write some pseudocode statements to add user-supplied strings to the bag until it becomes full.

```
1. // aBag is empty
   do
   {
      entry = next string read from user
      aBag.add(entry)
   } while (!aBag.isFull())
   // aBag is full
```

Question 3 Is it legal to have two versions of `remove`, one that has no parameter and one that has a parameter, in the same class? Explain.

Yes. The two methods have different signatures. They are overloaded methods.

Question 4 Given the full bag `aBag` that you created in Question 1, write some pseudocode statements that remove and display all of the strings in the bag.

```
// aBag is full
while (!aBag.isEmpty())
{
   entry = aBag.remove()
   Display entry
}
// aBag is empty
```

Checkpoint 2

Question 5 Given the full bag `aBag` that you created in Question 1, write some pseudocode statements to find the number of times, if any, that the string `"Hello"` occurs in `aBag`.

Question 6 Given the full bag `aBag` that you created in Question 1, write some Java statements that display all of the strings in `aBag`. Do not alter the contents of `aBag`.

Checkpoint 2

Question 5 Given the full bag `aBag` that you created in Question 1, write some pseudocode statements to find the number of times, if any, that the string `"Hello"` occurs in `aBag`.

```
Display "The string Hello occurs in aBag " + aBag.getFrequencyOf("Hello") + " times."
```

Question 6 Given the full bag `aBag` that you created in Question 1, write some Java statements that display all of the strings in `aBag`. Do not alter the contents of `aBag`.

```
String[] contents = aBag.toArray();
for (int index = 0; index < contents.length; index++)
   System.out.print(contents[index] + " ");
System.out.println();
```

Vending Machine Like An ADT

- Perform only available tasks
- User must understand the tasks
- Cannot access inside of mechanism
- Usable without knowing inside implementation
- Inside implementation unknown to users

Do Lab 2A Probs 1 & 2

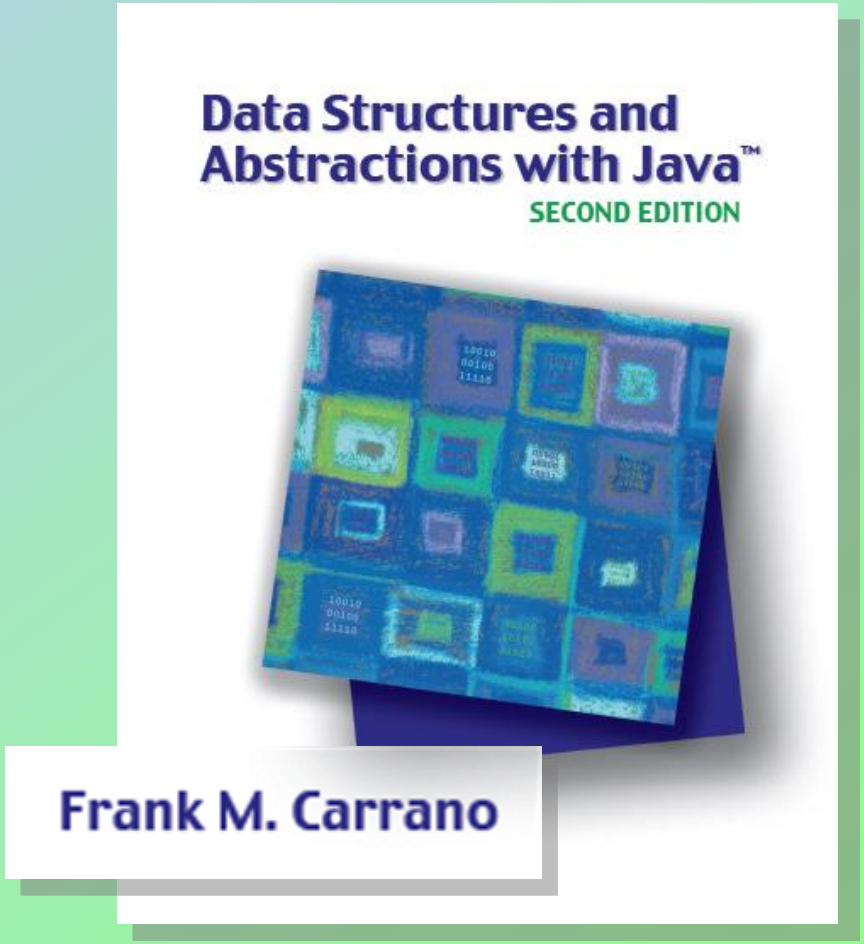
Java Collections has an official "Bag" interface, called Set

- The interface **Set (import java.util.Set)**
  - **a Set is a Bag that contains only unique entries**
  - **no duplicates**

```
public boolean add(Object newEntry)
public boolean remove(Object anEntry)
public void clear()
public boolean contains(Object anEntry)
public boolean isEmpty()
public int size()
public Object[] toArray()
```

Creating Classes from Other Classes

Appendix C



Slides by Steve Armstrong
LeTourneau University
Longview, TX

Appendix C Contents

- Composition
  - Generic Types
  - Adapters
- Inheritance ← we'll start here
  - Invoking Constructors from Within Constructors
  - Private Fields and Methods of The Base Class
  - Overriding, Overloading Methods
  - Multiple Inheritance
- Type Compatibility and Base Classes
  - The Class `Object`

Inheritance

- A general or base class is first defined
- Then a more specialized class is defined by …
  - Adding to details of the base class
  - Revising details of the more general class
- Advantages
  - Saves work
  - Common properties and behaviors are define only once for all classes involved

Inheritance

Embodies an "is a" relationship

A Wagon "is a" Vehicle

A PowerBoat "is a" Boat

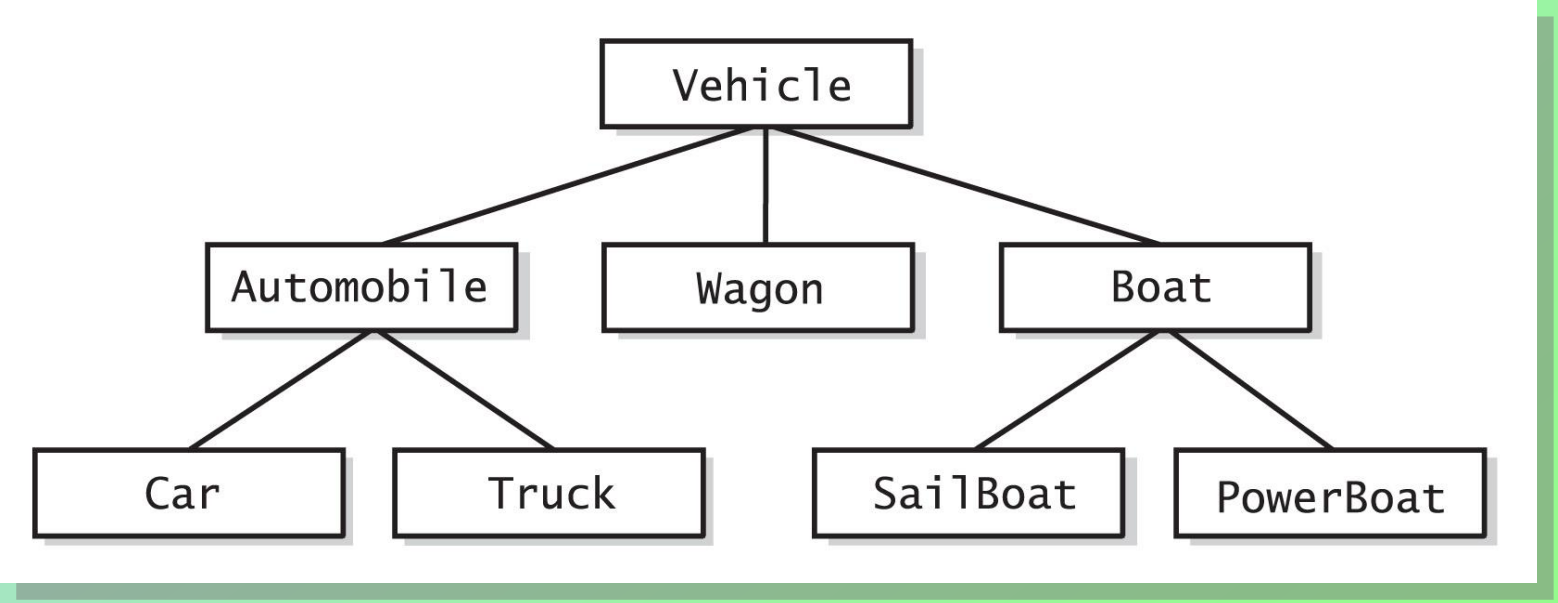


Fig. 2-2 A hierarchy of classes.

Inheritance

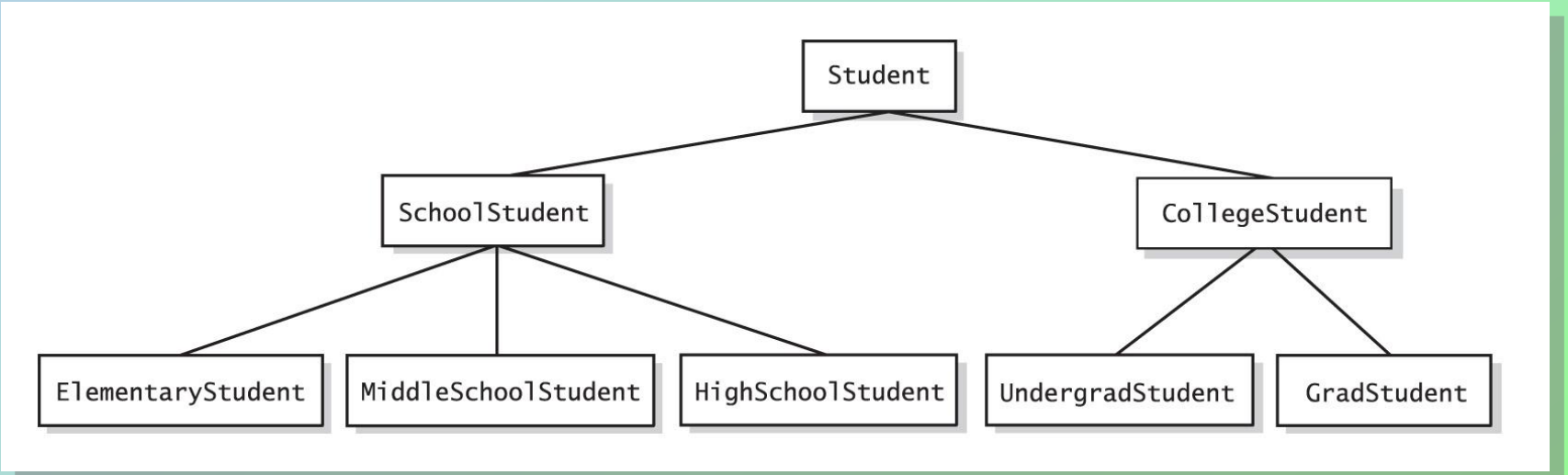


Fig. 2-3 A hierarchy of student classes.

Private Fields, Methods of Base Class

- Accessing inherited data fields
  - Not accessible by name within definition of a method from another class – including a derived class
  - Still they are inherited by the derived class
- Derived classes must use public methods of the base class
- Note that private methods in a base class are also unavailable to derived classes
  - But usually not a problem – private methods are used only for utility duties within their class

Inheritance

- The following slides will illustrate Inheritance by deriving a Student class from our Person class
  - A Student "is a" Person, with a studentNumber

- Notice the Student constructor, toString, equals method accesses the equivalent Person class method using super

```
public class Person
{
  private String name;


  public Person()          // default (or no-arg) constructor
  {
    name = "No name yet.";
  }

  public Person(String initialName)  // explicit constructor
  {
    name = initialName;
  }

  public void setName(String newName)  // modifier method
  {
    name = newName;
  }

  public String getName()   // accessor method
  {
    return name;
  }

  public void writeOutput()
  {
    System.out.println("Name: " + name);
  }

  public boolean sameName(Person otherPerson)
  {
    return (this.name.equalsIgnoreCase(otherPerson.name));
  }

}
```

Person Class

```
public class Student extends Person // Student inherits all code from Person class
{
  private int studentNumber;        // additional field for Student objects (in addition to name)

  public Student()    // no arg constructor
  {
    super();                    // *** invoke the no-arg constructor of the base class (Person)
    studentNumber = 0;  // *** set the studentNumber to 0, indicating no number yet
  }

  public Student(String initialName, int initialStudentNumber)
  {
    super(initialName);                                  // *** invoke the explicit constructor of the base class (Per
    studentNumber = initialStudentNumber;   // *** set the studentNumber to initialStudentNumber
  }

  public int getStudentNumber()
  {
    return studentNumber;
  }

  public void setStudentNumber(int newStudentNumber)
  {
     studentNumber = newStudentNumber;
  }

  public String toString()
  {
         return super.toString() + " " + studentNumber;  // *** invoke the base class toString and add studentNumber
  }

  public boolean equals(Object other)
  {
        if ( other instanceof Student )
        {
               Student that = (Student) other;
               return super.equals(that) && this.studentNumber == that.studentNumber;
        }      // *** check the name fields are the same using super.equals
        else
               return false;
  }
```

Student Class

refer to for Prob 3

Invoking Constructors from Within Constructors

- Constructors usually initialize data fields
- In a derived class
  - The constructor must call the base class constructor
- Note use of reserved word **`super`** as a name for the constructor of the base class
  - When **`super`** is used, it must be the first action in the derived constructor definition
  - Must not use the name of the constructor

Overriding Methods

- When a derived class defines a method with the same signature as in base class
  - Same name
  - Same return type
  - Same number, types of parameters
- Objects of the derived class that invoke the method will use the definition from the derived class
- It is possible to use **`super`** in the derived class to call an overridden method of the base class

Overriding Methods

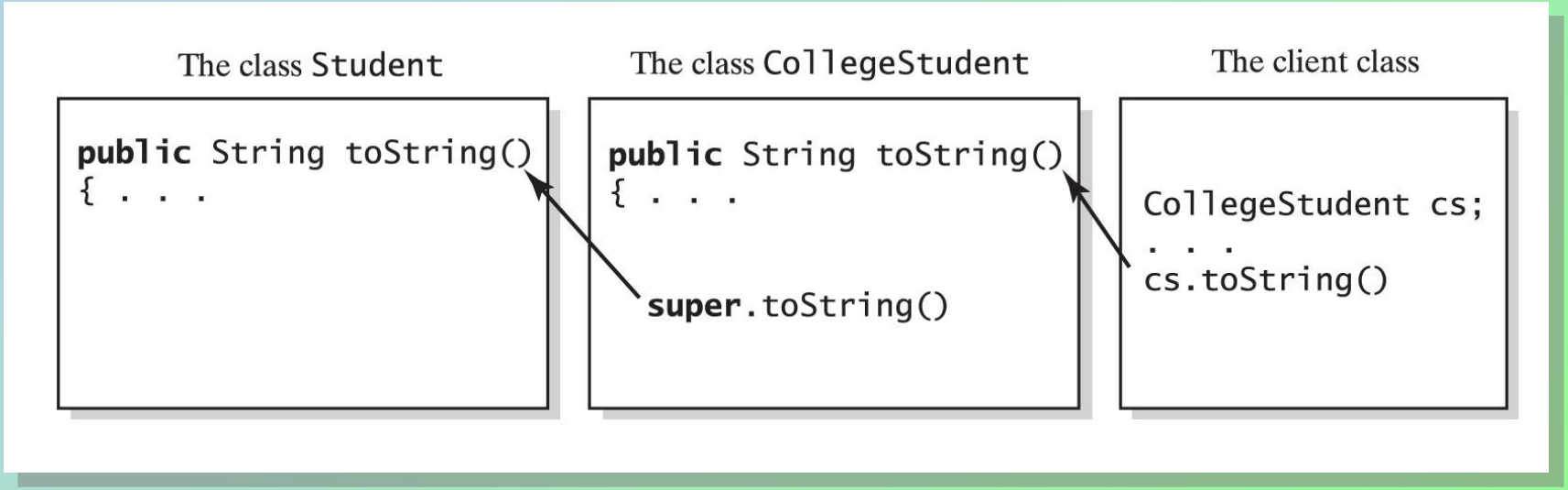


Fig. 2-5 The method **`toString`** in **`CollegeStudent`** overrides the method **`toString`** in **`Student`**

Overloading a Method

- When the derived class method has
  - The same name
  - The same return type … but …
  - Different number or type of parameters
- Then the derived class has available
  - The derived class method … and
  - The base class method with the same name
- Java distinguishes between the two methods due to the different parameters

Object Types of a Derived Class

- Given :
  - Class **`CollegeStudent`**,
  - Derived from class **`Student`**
- Then a **`CollegeStudent`** object is also a **`Student`** object
- In general …
  An object of a derived class is also an object of the base class

Question 10 If `HighSchoolStudent` is a subclass of Student, can you assign an object of `HighSchoolStudent` to a variable of type Student? Why or why not?

Question 11 Can you assign an object of `Student` to a variable of type `HighSchoolStudent`? Why or why not?

10.

Yes. You can assign an object of a class to a variable of any ancestor type. An object of type `HighSchoolStudent` can do anything that an object of type Student can do.

11.

No. The `Student` object does not have all the behaviors expected of a `HighSchoolStudent` object.

The Class `Object`

- Every class is a descendant of the class `Object`
- `Object` is the class that is the beginning of every chain of derived classes
  - It is the ancestor of every other class
  - Even those defined by the programmer

- http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Object.html

Speaking of class Object...

Ancestor of all classes

- defines methods clone(), equals(), toString(), among others
- all classes automatically derive these methods from Object
- to be useful, we have to override them so they work with the details of the class in which they are defined

Overriding the equals method

the parameter is class Object, we use a *cast* to be able to refer to it as a Name or Student or BankAccount or whater class we are defining it for.

we can refer to the private data members of `that` since it is an object of the same class. Here is equals for Name:

```
public boolean equals (Object other){
  if (other instanceof Name){
    Name that = (Name) other;
    return  this.first.equals(that.first) &&
            this.last.equals(that.last)
  }
  else return false;
}
```

Why equals(Object other) ?

- http://stackoverflow.com/questions/12787947/overriding-object-equals-vs-overloading-it

Overriding versus Overloading
Let's start with the difference between overriding and overloading. With overriding, you actually *redefine* the method. You remove its original implementation and actually replace it with your own. So when you do:

```
@Override public boolean equals(Object o)
    { ... }
```

You're actually re-linking your new equals implementation to replace the one from Object (or whatever superclass that last defined it).
On the other hand, when you do:

```
public boolean equals(MyClass m)
{ ... }
```

You're defining an entirely new method because you're defining a method with the same name, but different parameters. When contains calls equals, it essentially calls it on a variable of the type Object.

Question 12 If sue and susan are two instances of the class Name, what if statement can decide whether they represent the same name?

12.

```
if (sue.equals(susan))
```

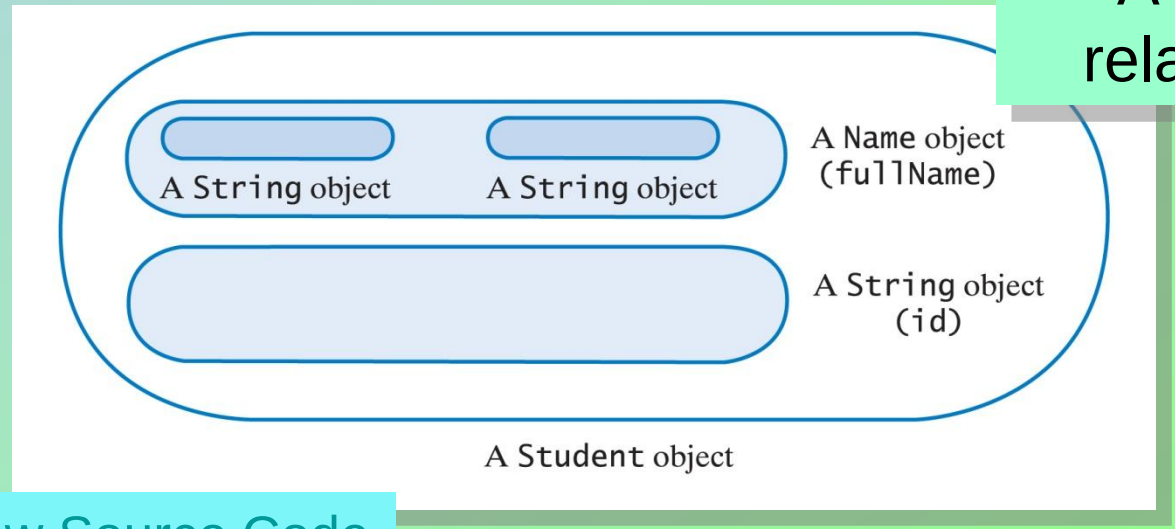
Inheritance vs Composition

- Inheritance is only one way to make classes from other classes
  - Embodies "is-a" relationship between classes
  - If "is-a" doesn't apply then can't inherit
- Composition is another way
  - Objects of one class are composed of objects of one or more other classes.
  - Embodies a "has-a" relationship

Composition

- When a class has a data field that is an instance of another class
- Example – an object of type **`Student`**.

A "has a" relationship



Click to View Source Code

Fig. 2-1 A **`Student`** object composed of other objects

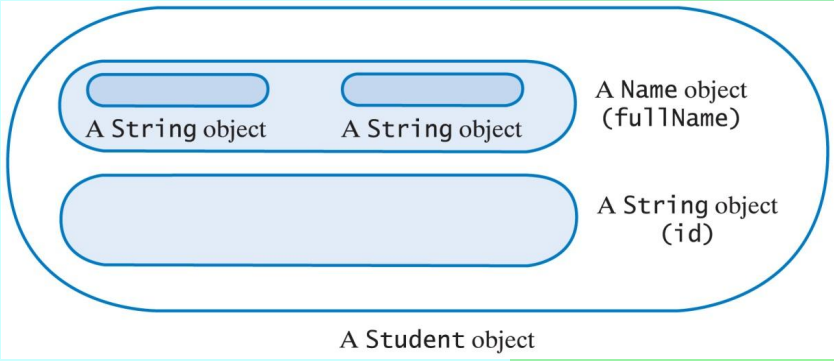
```
public class Student
{
    private  Name fullName;
    private  String id; // identification number


    public Student ()
    {
        fullName = new Name ();
        id = "";
    } // end default constructor


    public Student (Name studentName, String studentId)
    {
        fullName = studentName;
        id = studentId;
    } // end constructor


    public void setStudent (Name studentName, String studentId)
    {
        setName (studentName); // or fullName = studentName;
        setId (studentId); // or id = studentId;
    } // end setStudent


    public void setName (Name studentName)
```



Adapters

- Use composition to write a new class
  - Has an instance of an existing class as a data field
  - Defines new methods needed for the new class
- Example – a **`NickName`** class adapted from class **`Name`**
  - **`Inside a NickName object is a Name object to hold the data`**

Name Class

```
public class Name
{
   private String first; // first name
   private String last; // last name

   public Name ()
   {
      first = "";
      last = "";
   } // end default constructor

   public void setFirst (String firstName)
   {
      first = firstName;
   } // end set First

   public String getFirst ()
   {
      return first;
   } // end getFirst

   public void setLast (String lastName)
   {
      last = lastName;
   } // end setLast

   public String getLast ()
   {
      return last;
   } // end getLast
} // end class Name
```

NickName Class
Adapter of Name class

```
public class NickName
    {
       private Name nick;
       public NickName ()
       {
          nick = new Name ();
       } // end default constructor

       public void setNickName (String nickName)
       {
          nick.setFirst (nickName);
       } // end setNickName

       public String getNickName ()
       {
          return nick.getFirst ();
       } // end getNickName
    } // end NickName
```

Question 5 Write statements that define bob as an instance of NickName to represent the nickname Bob. Then, using bob, write a statement that displays Bob.

5.

```
NickName bob = new NickName();
bob.setNickName("Bob");
System.out.println(bob.getNickName());
```

Lab 2A Problem 4

- To illustrate the use of Adapter classes, we can make a Club class as an adapter for a Bag<Person> objects.
  - Avoids the awkwardness of
    - Bag<Person> CS_club = new Bag<Person>();
  - Lets us write instead
    - Club CS_club = new Club();

Setting up the Club class

- class Club as an adapter of class Bag<Person>

```
public class Club
{
        private  Bag<Person>  members;  // the set of club members
```

- The default constructor just creates a new Bag

```
public Club()
{
        members = new Bag<Person>() ;
}
```

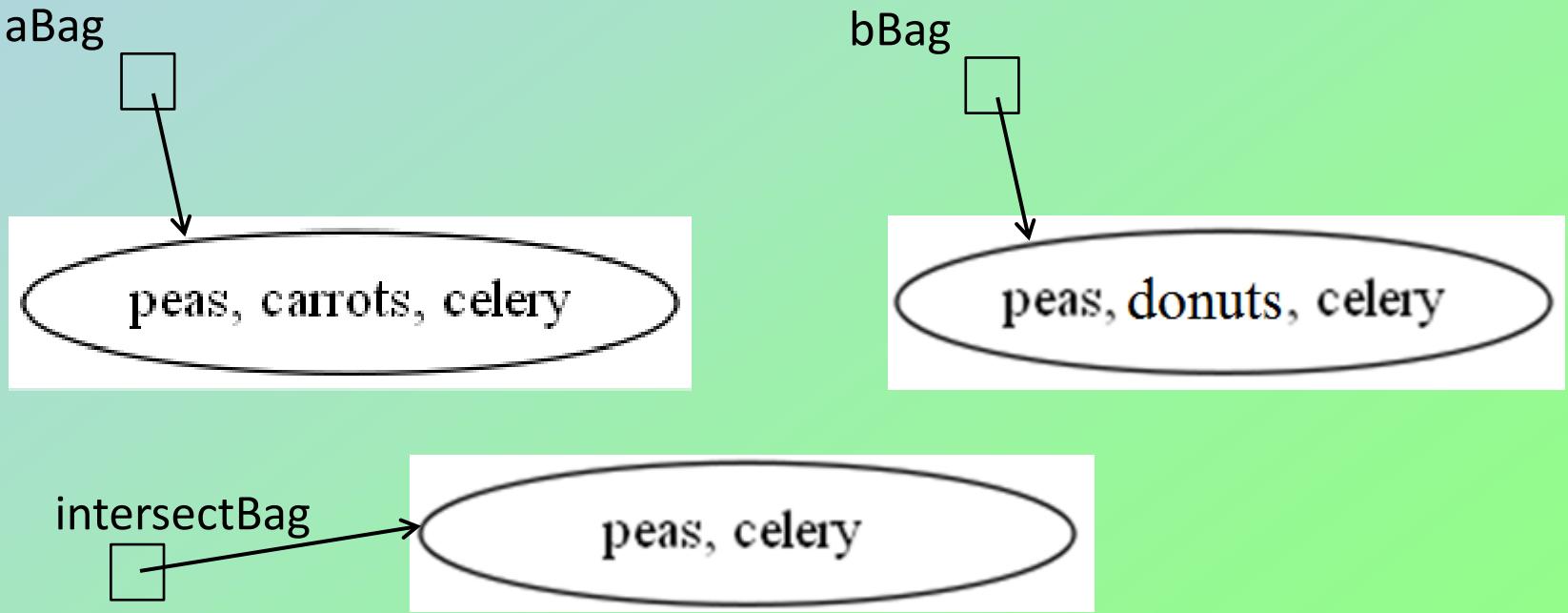
Other Club class methods

- Most Club class methods will just directly call methods on the Bag instance variable

```
public boolean isFull()
{
        return members.isFull();
}
```

Work on Lab 2A Problems 3 and 4

- Challenge problem: write an intersection method to compute the items in common between two Bag objects.



End

Chapter 1