Data Structures with Java and JUnit

Collection Considerations

Chapter 13
© Rick Mercer

Outline

- Consider a Bag Abstract Data Type
- Java Interfaces
    - Method headings only, must be implemented by a Java class (or two or many)
- Data Structures
- Collections Classes
- Generics
    - With Object parameters
    - With type parameters <T>

Some Definitions

Abstract Data Type (ADT) A set of data values and associated operations that are precisely specified independent of any particular implementation.

Bag, Set, List, Stack, Queue, Map

Collection Class A Java language construct for encapsulating the data and operations

ArrayList, LinkedList, Stack, TreeSet, HashMap

Data Structure An organization of information usually in memory

arrays, linked structure, binary trees, hash tables

Why Collection classes?

- Need collections to store data to model real world entities
  - All courses taken by one student
  - All students in a class
  - A set of Poker Hands to simulate Texas Hold'em game
  - An appointment book
  - List of things on your cell phone
  - The movies in your movie queue

Common Methods

- Collection classes often have methods for performing operations such as these
  - Adding an object to the collection of objects
  - Removing an object from the collection
  - Getting a reference to a particular object *find*
    - then you can send messages to the object while it is till in the collection. Program do this a lot
  - Retrieving certain objects such as the most recently pushed (Stack) or least recently enqueued (Queue)
  - Arranging objects in a certain order *sorting*
- The most basic collection is a Bag

NIST Definition of the Bag

http://xlinux.nist.gov/dads/HTML/bag.html

Bag

Definition: Unordered collection of values that may have duplicates

Formal Definition: A bag has a single query function, occurencesOf(v, B), which tells how many copies of an element are in the bag, and two modifier functions, add(v, B) and remove(v, B). These may be defined with axiomatic semantics as follows.

```
1. new() returns a bag
2. occurencesOf(v, new()) = 0
3. occurencesOf(v, add(v, B)) = 1 + occurencesOf(v, B)
4. occurencesOf(v, add(u, B)) = occurencesOf(v, B) if v ≠ u
5. remove(v, new()) = new()
6. remove(v, add(v, B)) = B
7. remove(v, add(u, B)) = add(u, remove(v, B)) if v ≠ u
      where B is a bag and u and v are elements.
```

isEmpty(B) may be defined with the following additional axioms:

```
8. isEmpty(new()) = true
9. isEmpty(add(v, B)) = false
```

Also known as multi-set.

We use Java interfaces rather than axiomatic expressions

```
/**
 * This interface specifies the methods for a Bag ADT. A bag
 * is also known as a multi-set because bags are like a set
 * with duplicate elements allowed.
 */
public interface Bag {
  // Return true if there are no elements in this Bag.
  public boolean isEmpty();

  // Add a v to this collection.
  public void add(Object v);

  // Return how often the value v exists in this StringBag.
  public int occurencesOf(Object v);

  // If an element that equals v exists, remove one
  // occurrence of it from this Bag and return true.
  // If occurencesOf(v) == 0, simply return false.
  public boolean remove(Object v);
}
```

The Java interface construct

- A Java **`interface`** describes a set of methods:
  - no constructors
  - no instance variables
- The interface must be implemented by some class.
  - Over 1,000 java classes implement one or more interfaces
- Consider a simple **`interface`**

interface BarnyardAnimal

```
public interface BarnyardAnimal  {
  public String sound( );
}
```

- Interfaces have public method headings followed by semicolons.
  - no { } *static methods are allowed but rare*
- No methods are implemented
  - One or more classes implement the methods

Classes implement interfaces

- To implement an interface, you must have all methods as written in the interface

```
public class Cow implements BarnyardAnimal {
  public String sound() {
    return "moo";
  }
}

public class Chicken implements BarnyardAnimal {
  public String sound() {
    return "cluck";
  }
}
```

`Cow` *and* **`Chicken`** *are also known as a* **`BarnyardAnimal`**

```
BarnyardAnimal aCow = new Cow();
BarnyardAnimal aChicken = new Chicken();

assertEquals(________, aCow.sound());

assertEquals(________, aChicken.sound());
```

- Fill in the blanks so the assertions pass
- We can store references to a `Cow` and a `Chicken` into reference variable of type `BarnyardAnimal`

Comparable interface (less silly)

- Can assign an instance of a class that implements an **`interface`** to a variable of the **`interface`** type

```
Comparable str = new String("abc");
Comparable acct = new BankAccount("B", 1);
Comparable day = new Date();
```

- A few classes that **`implement`** `Comparable`

```
BigDecimal BigInteger  Byte  ByteBuffer
Character  CharBuffer  Charset  CollationKey
Date  Double  DoubleBuffer  File  Float
FloatBuffer  IntBuffer  Integer  Long
LongBuffer  ObjectStreamField  Short
ShortBuffer  String  URI
```

- Comparable defines the "natural ordering"

  When is one object less than or greater than another?

Implementing Comparable

- Any type can implement Comparable to determine if one object is less than, equal or greater than another

```
public interface Comparable<T>  {
/**
   * Return 0 if two objects are equal; less than
   * zero if this object is smaller; greater than
   * zero if this object is larger.
   */
   public int compareTo(T other);
}
```

Let BankAccount be Comparable

```
public class BankAccount implements Comparable<BankAccount> {
  private String ID;
  private double balance;

  public BankAccount(String ID...

    // stuff deleted


  public int compareTo(BankAccount other) {
    // Must complete this method or else it's an error.
    // Compare by ID, might as well use String's compareTo
    return getID().compareTo(other.getID());
  }
}
```

guarantee this class has a **`compareTo`**

Add this method

A test method—IDs are compared

```
@Test
public void testCompareTo() {
  BankAccount a = new BankAccount("Alice", 543.21);
  BankAccount z = new BankAccount("Zac", 123.45);
  assertTrue(a.compareTo(a) == 0);
  assertTrue(z.compareTo(z) == 0);
  assertTrue(a.compareTo(z) < 0);
  assertTrue(z.compareTo(a) > 0);
  assertTrue(a.compareTo(z) <= 0);
  assertTrue(z.compareTo(a) >= 0);
  assertTrue(z.compareTo(a) != 0);
  assertTrue(a.compareTo(z) != 0);
}
```

Generic Collections Classes

1) With Object parameters
2) With Java generics using type parameters

Outline

- Class **`Object`**, casting, and a little inheritance
- Generic Collections with Object[]
- Using <Type> to give us type safety
- Autoboxing / Unboxing

Can have one Collection class for any type

```
public class ArrayBag implements Bag {
  // --Instance variables
  private Object[] data;
  private int n;

  // Construct an empty bag that can store any type
  public ArrayBag() {
    data = new Object[20];
    n = 0;
  }

  public void add(Object element) {  }
  public int occurencesOf(Object element) {  }
  public boolean remove(Object element) {  }
}
```

What was that **`Object`** *thing?*

- Java has a class named `Object`
- It communicates with the operating system to allocate memory at runtime
- `Object` has 11 methods
- `Object` is the *superclass* of all other classes
  - All classes extend `Object` or a class that `extends Object`, or a class that extends a class that `extends Object`, or …

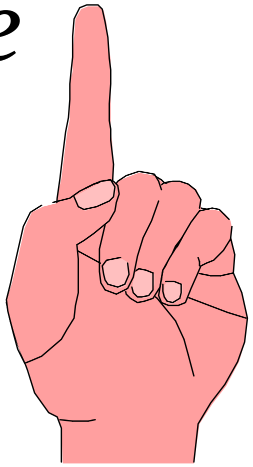
`EmptyClass` *inherits all 11 methods defined in class Object*

```
public class EmptyClass extends Object {
   // This class inherits Object's 11 methods
}

// Inherits 11 methods from Object
EmptyClass one = new EmptyClass();
EmptyClass two = new EmptyClass();
System.out.println(one.toString());
System.out.println(one.hashCode());
System.out.println(one.getClass());
System.out.println(two.toString());
System.out.println(two.hashCode());
System.out.println(one.equals(two));
one = two;
System.out.println(one.equals(two));
```

Output

```
EmptyClass@ffb8f763
-4655261
class EmptyClass
EmptyClass@ffbcf763
-4393117
false

true
```

One way assignment: up the hierarchy, but not down



- Can assign any reference to an **`Object`** object

```
Object obj1 = new String("a string");
Object obj2 = new Integer(123);
System.out.println(obj1.toString());
System.out.println(obj2.toString());
```

Output

```
a string
123
```

- But not the other way *compiletime error*

```
String str = obj1;  // incompatible types
           ^
```

Type mismatch: cannot convert from Object to String

Tricking the compiler into believing **`obj1`** *is* **`String`**

- Sometimes an explicit cast is needed
  - Enclose the class name with what you know the class to be in parentheses (`String`) and place it before the reference to the `Object` object.

$$\texttt{str = (String)}\underbrace{\texttt{obj1}}_{\text{A reference to an } \mathbf{\texttt{Object}} \text{ object}}\texttt{;}$$

Example Casts

```
Object obj1 = new String("A string");
String str = (String) obj1;

Object obj2 = new Integer(123);
Integer anInt = (Integer) obj2;

Object obj3 = new Double(123.45);
Double aDouble = (Double) obj3;

Object obj4 = new BankAccount(str, aDouble.doubleValue());
Double balance = ((BankAccount) obj4).getBalance();
```

ClassCastException

- Does this code compile by itself?

```
Object obj3 = new Double(123.45);
String aString = (String) obj3;
```

- Does that code run?
- Let's apply all of use of **`Object`** to one collection class that can store any type (see next slide)

`Object` *can store a reference to type*

- The `Object` class allows collections of any type
- Use `Object[]` rather than any one specific type

```
public class ArrayBag implements Bag {

  private Object[] data;
  private int n;

  public ArrayBag() {
    data = new Object[20];
    n = 0;
  }

  public void add(Object element) {
```

Object as a parameter and a return type

- `add` has an `Object` parameter
- `get` has an `Object` and return type
- This means that you add or retrieve references to any type object *even primitives as we'll see later*
- This is possible because of *inheritance*
  - can to assign a reference to `Object`
  - All Java classes extend `Object`

One class for many types

but oh that ugly cast …

```
Bag names = new ArrayBag();
names.add("Kim");
names.add("Devon");
// cast required
String element = (String)names.get(0);

GenericArrayBag accounts = new GenericArrayBag();
accounts.add(new BankAccount("Kim", 100.00));
accounts.add(new BankAccount("Devon", 200.00));
// cast required
BankAccount current= (BankAccount)accounts.get(1);
```

Generics via Type Parameters <E>

A better way to implement collection classes

Assumption: GenericArrayBag has been implemented using an Object[] instance variable, an Object parameter in **`add`**, and an Object return type in **`get`**

One Problem with the old way using Object parameters and return types

- Java "raw" types (no generics) do not check the type
- This is legal code

```
 Bag name = new ArrayBag();
names.add(new Integer(2));
names.add(new BankAccount("Pat", 2.00));
names.add(new GregorianCalendar(2009, 0, 1));
names.add(1.23);
```

- So what type do you promise the compiler for these expressions?

```
names.get(0) ______
names.get(1) ______
names.get(2) ______
names.get(3) ______
```

- Often get the runtime error **`ClassCastException`**
- With version 5, Java added a better option: Generics

Generics

- Java 5 introduced Generics for type safety
  - specify the type of element to be added or returned
- Reference types are passed as arguments between **< >**

```
Bag<String> strings = new ArrayBag<String>();
Bag<Integer> ints = new ArrayBag <Integer>();
Bag<BankAccount> accounts = new ArrayBag <BankAccount>();
```

- Change the class heading and the compiler sees **`E`** (or any identifier you use) as the argument type used during construction **`E`** *could represent String, Integer, BankAccount, …*

```
public class ArrayBag<E> implements Bag<E>
```

Can't add the wrong type

- Java generics checks the type at compile time
  - See errors early--a good thing
  - Known as "type safety" because you can't add different types

```
Bag<String> strings = new ArrayBag<String>();
strings.add("Pat");                           // Okay
Strings.add(new BankAccount("Pat", 12));   // Error

ArrayBag <Integer> ints = new ArrayBag <Integer>();
int.add(1);                     // Okay
int.add(new String("Pat")); // Compiletime Error
```

Type parameter <E>

- Type parameters *the new way to have a generic collection*

```
public class ArrayBag<E> implements Bag<E> {

  private Object[] data;
  private int n;

  public ArrayBag() {
    data = new Object[20];
    n = 0;
  }

  public void add(E element) { ... }


  public E get(int index) { ... }
```

Can not have **E[]**

- We can not declare arrays of a generic parameter

```
public class ArrayBag<E> implements Bag<E> {

  private E[] data;

  public ArrayBag() {
    data = new E[1000];
                ^
```

Cannot create a generic array of E

Can cast with (Type[])

- At runtime, the generic parameter `E` is really `Object`
- We *could* use a cast like this: **`(E[])`**

```
public class ArrayBag<E> implements Bag<E> {

   private E[] data;

   public ArrayList() {
     data = (E[]) (new Object[1000]);
   }
```

Use Object[]

- Or we can use `Object[]` as the instance variable

```
public class ArrayBag<E> implements Bag<E> {

  private Object[] data;

  public ArrayBag() {
    data = new Object[1000];
  }
```

Another Advantage:

We can "appear" to add primitives

- Using this method heading

```
  public void add(E element) { ... }
```

- How can this code be legal?

```
ArrayBag<Integer> ints = new ArrayBag <Integer>();

ints.add(new Integer(5));
ints.add(5);  // 5 is int, not an Integer
```

Primitives are appear to be Objects

- To allow collections of primitive types, with **`Object[]`** Java provided wrapper classes:

  `Integer Double Character Boolean Long Float`

  These allow you to treat primitives as Objects (need **`new`**)

- Before Java 5, wrapper objects can't handle arithmetic operators

```
Integer anInt = new Integer(50);
int result = 2 * anInt - 3;
```

↖ Error before Java 5

- Now objects *appear* as primitives and primitives as objects

```
Integer i = 3; // assign int to Integer
int j = new Integer(4); // Integer to int
int k = (3*new Integer(7)) + (4*i); // operators OK
```

How? Boxing / Unboxing

- Autoboxing is the process of treating a primitive as if it were an reference type. When the compiler sees this

```
Integer anInt = 3;
```

  – the code transforms into this

```
Integer anInt = new Integer(3);
```

- Java 1.4 arithmetic requires `intValue` or `doubleValue` messages, for example:

```
   int answer = 2 * anInt.intValue();
```

- Java 5 allows this

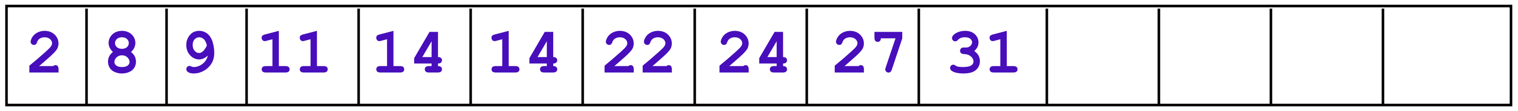
```
   int answer = 2 * anInt;
```

Collection Classes

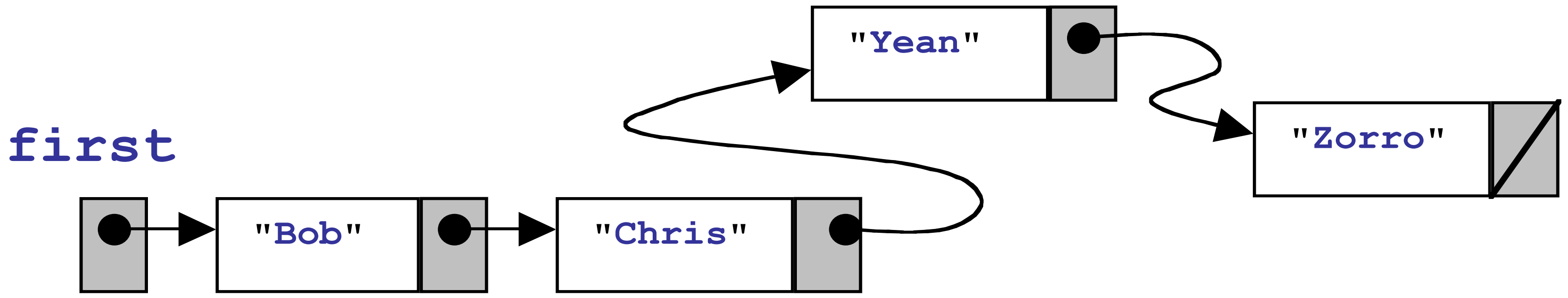
Structures to store elements

- Collection classes store data in many ways, here are 4

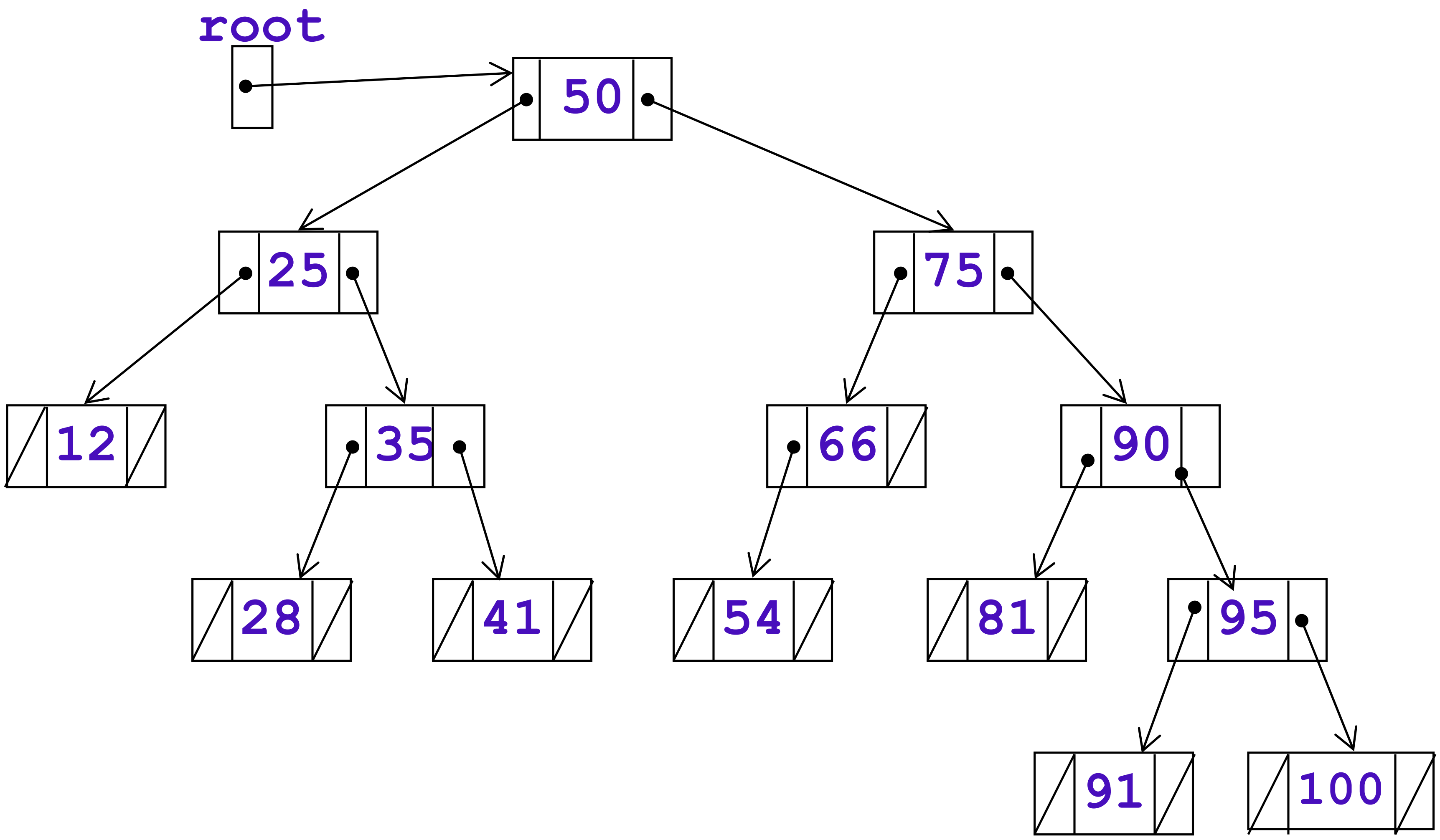
  1) in contiguous memory (arrays)



  2) or in a singly linked structure



3) or in a hierarchal structure such as a tree



4) or in hash tables

- Maps associate a key with a value
  - e.g. your student ID and your student record
- Elements could be stored in a hash table

Array Index	**Key**	**Object (state is shown, which is the instance variables values of Employees**
0	`Smith D`	`Devon 40.0 10.50 1 'S'`
1	`null`	`null`
2	`Gupta C`	`Chris 0.0 13.50 1 'S'`
3	`Herrs A`	`Ali 20.0 9.50 0 'S'`
4	`null`	`null`
5	`Li X`	`Xuxu 42.5 12.00 2 'M'`