Using the Java Collection Libraries Lecture 2

- What support is offered by Java for programming with data structure?

- What type of data structure can be created using Java?
- How do we create and access data structure under Java?

- Some real examples?

Menu

- Overview of Data Structure Programming Topics
- Programming with Libraries
- Collections
- Programming with Lists of Objects

Overview of Data Structure Programming in this Course: Part 1

Programming with **Linear** collections

- Kinds of collections:
  - Lists, Sets, Bags, Maps, Stacks, Queues, Priority Queues
- Using Linear collections
  - Programming with collections
  - Searching & Sorting Data
  - Implementing linear collections
  - Implementing sorting algorithms
  - Linked data structures and “pointers”

Overview of Data Structure Programming in this Course: Part 2

Programming with **Hierarchical** collections

- Kinds of collections:
  - Trees, binary trees, general trees
- Using tree structured collections
  - Building tree structures
  - Searching tree structures
  - Traversing tree structures
- Implementing tree structured collections
- Implementing linear collections
  - with binary search trees

Programming with Libraries

- Modern programs (especially GUI and network) are too big to build from scratch.

  ⇒ Have to reuse code written by other people

- Libraries are collections of code designed for reuse.
  - Java has a huge collection of standard libraries….
    - Packages, which are collections of
      - Classes
  - There are lots of other libraries as well
- Learning to use libraries is essential.

Java API

- What are the benefits of reuse?

Libraries to use

- java.util — **Collection** classes; Other **utility** classes
- java.io — Classes for **input and output**
- javax.swing, java.awt — Large library of classes for **GUI** programs

We will use these libraries in almost every program

Using Libraries

- Read the documentation to pick useful library

Java API

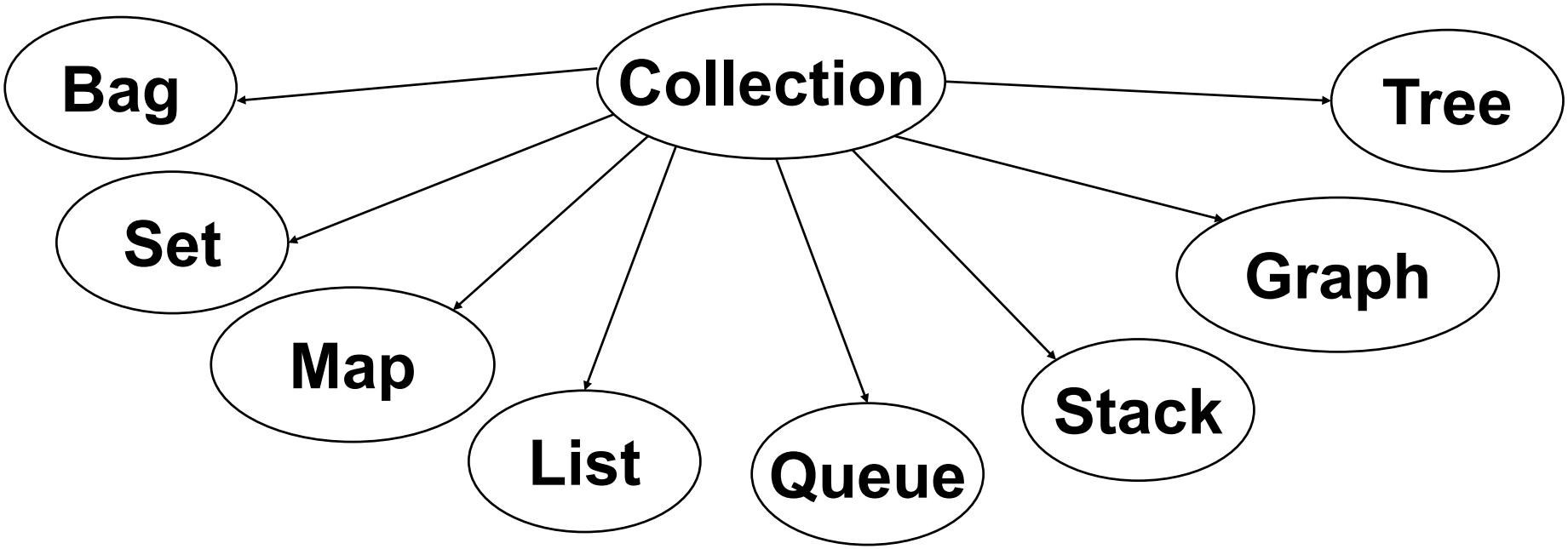
- **import** the package or class into your program

```
import java.util.*;
import java.io.*;
```

- Read the documentation to identify how to use
  - **Constructors** for making instances
  - **Methods** to call
  - **Interfaces** to implement
- Use the classes as if they were part of your program

“Standard” Collections

Common ways of organising a collection of values:



Each of these is a different **type** of collection

- values organised/structured differently
- different constraints on duplicates, and on access
- very abstract –
  - don’t care what type the elements are.
  - don’t care how they’re stored or manipulated inside

Abstract Data Types

Set, Bag, Queue, List, Stack, Map, etc are

Abstract Data Types (outcome of abstraction / encapsulation)

- an ADT is a type of data, described at an abstract level:
  - Specifies the **operations** that can be done to an object of this type
  - Specifies how it will **behave**.
- eg Set: (simple version)
  - Operations: **add(***value* **)**, **remove(***value* **)**, **contains(***value* **)**→*boolean*
  - Behaviour:
    - A new set contains no values.
    - A set will contain a value *iff*
      - the value has been added to the set and
      - it has not been removed since adding it.
    - A set will not contain a value *iff*
      - the value has never been added to the set, or
      - it has been removed from the set and has not been added since it was removed.

Java Collections library

Interfaces:

- *Collection*
  = Bag (most general)
- *List*
  = ordered collection
- *Set*
  = unordered, no duplicates
- *Queue*
  ordered collection, limited access
  (add at one end, remove from other)
- *Map*
  = key-value pairs (or mapping)

Classes

- List classes:
  ArrayList, LinkedList, vector…
- Set classes:
  HashSet, TreeSet,…
- Map classes:
  HashMap, TreeMap, …
- …

Java Interfaces and ADT's

- A Java ***Interface*** corresponds to an Abstract Data Type
  - Specifies what methods can be called on objects of this type (specifies name, parameters and types, and type of return value)
  - Behaviour of methods is only given in comments (but cannot be enforced)

× No constructors - can't make an instance: **new** Set()

× No fields - doesn't say how to store the data

× No method bodies. - doesn't say how to perform the operations

---

```
public interface Set {
    public void add(??? item);                  /*…description…*/
    public void remove(??? item);               /*…description…*/
    public boolean contains(??? item);          /*…description…*/
    …
        // (plus lots more methods in the Java Set interface)
```

List Interface in Java

The real **List** interface in Java 1.5 is defined as follows.

```
public interface List<E> extends Collection<E> {
...
boolean add(E o);
E get(int index);
...
boolean contains(Object o);
Iterator<E> iterator(); ...
}
```

Using Java Collection Interfaces

- Your program can
  - Declare a variable, parameter, or field of the interface type

```
private List drawing;          // a list of Shapes
```

  - Call methods on that variable, parameter, or field

```
drawing.add(new Rect(100, 100, 20, 30))
```

× Problem:

  - How do we specify the type of the values?

Parameterised Types

- The structure and access discipline of a collection is the same, regardless of the type of value in it:
  - A set of Strings, a set of Persons, a set of Shapes, a set of integers all behave the same way.

⇒ Only want one Interface for each kind of collection.

(there is only one *Set* interface)

- Need to specify kind of values in a particular collection

⇒ The collection Interfaces (and classes) are parameterised:

- Interface has a **type parameter**
- When declaring a variable collection, you specify
  - the type of the collection and
  - the type of the elements of the collection

Parameterised Types

- Interfaces may have type parameters (*eg*, type of the element):

It's a Set of something, as yet unspecified

```
public interface Set <T> {
    public void add(T item);          /*…description…*/
    public void remove(T item);       /*…description…*/
    public boolean contains(T item); /*…description…*/
    …   // (lots more methods in the Java Set interface)
```

- When declaring variable, specify the actual type of element

```
private Set <Person>  friends;
private List <Shape> drawing;
```

Collection Type

Type of value in Collection

Using the Java Collection Library

Problem:

- How do you create an instance of the interface?

  Interfaces don't have constructors!

  ```
  private List <Shape> drawing = new ???? ( );
  ```

- **Classes** in the Java Collection Library ***implement*** the interfaces
  - Define constructors to construct new instances
  - Define method bodies for performing the operations
  - Define fields to store the values

  ⇒ Your program can create an instance of a class.

  ```
  private List <Shape> drawing = new ArrayList <Shape> ( );

  Set <Person> friends = new HashSet <Person> ( );
  ```

ArrayList

- Part of the Java **Collections** framework.
  - predefined class
  - stores a list of items,
    - a collection of items kept in a particular order.
  - part of the java.util package

    ⇒ need to **import java.util.*;** at head of file

- You can make a new ArrayList object, and put items in it
  - Don't have to specify its size
  - Should specify the type of items.
    - new syntax: "type parameters"
  - Like an infinitely stretchable array
  - But, you can't use the […] notation
  - you have to call methods to access and assign

Using ArrayList: declaring

List of students

- Array:

```
private static final int maxStudents = 1000;
private Student[ ] students = new Student[maxStudents];
private int count = 0;
```

- Alternatively, we can do the following…
- ArrayList:

```
private ArrayList <Student> students = new ArrayList <Student>();
```

  - The type of values in the list is between "<" and ">" after ArrayList.
  - No maximum; no initial size; no explicit count

Using ArrayList: methods

- ArrayList has many methods! , including:
  - size(): returns the number of items in the list
  - add(*item*): adds an item to the *end* of the list
  - add(*index, item*): inserts an item at *index* (relocates later items)
  - set(*index, item*): replaces the item at *index* with *item*
  - contains(*item*): true if the list contains an item that equals *item*
  - get(*index*): returns the item at position *index*
  - remove(*item*): removes an occurrence of item

    (what if there are duplicates in the ArrayList?)
  - remove(*index*): removes the item at position *index*

    (both relocate later items)

  - You can use the “for each” loop on an array list, as well as a **for** loop

Using ArrayList

```
private ArrayList <Student> students = new ArrayList <Student>();
        :

Student s = new Student("Lindsay King", "300012345")
students.add(s);
students.add(0, new Student(fscanner));

for (int i = 0; i<students.size(); i++)
    System.out.println(students.get(i).toString());

for (Student  st : students)
    System.out.println(st.toString());

if (students.contains(current)){
    file.println(current);
    students.remove(current);
}
```

Q&A

- Name 3 type of collections that can be implemented under Java Programming with Linear collections
- Name 3 operations that can be implemented under Java Programming with Linear collections
- Name 2 type of collections that can be implemented under Java Programming with Hierarchical collections
- Name 4 operations that can be implemented under Java Programming with Hierarchical collections
- What is a software “library”?
- Define Java “Package”.
- Name Java’s IO library.
- Name Java’s GUI library.
- What is the Java statement to include package or class into your program?

Conclusions

- We can declare the type of a variable/field/parameter to be a collection of some element type
- We can construct a new object of an appropriate collection class.

What's next?

- What can we do with them?
  - What methods can we call on them?
  - How do we iterate down all the elements of a collection?
- How do we choose the right collection interface and class?

Readings

- [Mar07] Read 3.3
- [Mar13] Read 3.3