Generics and Abstract Data Types

Chapters 16 and 20

Collections

- A *collection* is an object that holds other objects
- A collection provides services for managing the elements it contains
  - Adding elements
  - Removing elements
- Collections can:
  - Be ordered or unordered
  - Be homogeneous (containing all the same type) or heterogeneous (containing different types)
  - Allow or disallow duplicates

Java Collections

- Implement the Collection interface
  - add
  - remove
  - isEmpty
  - iterator
- Set (no duplicates)
- List (ordered)
- Map (keys and values)
- Queue (first-in, first-out)
- Stack (first-in, last-out)

The Collections Class

- Provides static methods such search, sort, and shuffle that can be applied to various Collection objects
- (Note Collection vs. Collections!)

Lists

- A list is ordered Collection that is allowed to contain duplicates.
  - Lists can be heterogeneous or, through the use of generics, homogeneous.
  - Indices start at zero.
  - Lists allow you to manipulate elements through with an index
- ArrayList and LinkedList implement the List interface
  - ArrayLists are more efficient at accessing elements
  - LinkedLists are more efficient at adding an element to the beginning of a list (or in the middle if the add happens when you are iterating through the list)

Practice

- Create a Course object that is described by a course name, capacity, and a list of students.
  - A student is described by a name, id, and a status of whether the tuition is paid.
- Add methods:
  - addStudent
  - isEnrolled
  - dropStudent
  - dropAllUnpaidStudents
  - toString should include a print of the sorted roster

Stacks

- First-in, last-out
  - Last-in, first-out
  - Example: plates in a dispenser
- Many uses in programming
  - Java runtime stack (method calls)
  - Checking balanced parentheses/brackets
  - Evaluating expressions
- Methods
  - push
  - pop
  - peek
  - isEmpty

Practice

- Trace:

  push(“Alice”)

  push(“Bob”)

  peek

  pop

  push("Dave")

  pop

  peek

- Write a method to determine if a String is a palindrome (the same forwards and back). Use a stack.

Balancing Parentheses

- Compilers use stacks frequently.
- One use is to determine whether parentheses or brackets are properly matched up.
- To check this, we can ignore all other values in the expression and just look at the parentheses.
- There are four cases:
  - balanced — example: { [( )] }
  - unbalanced- extra open parenthesis — example: { ( )
  - unbalanced- extra closed parenthesis — example: ( ) }
  - unbalanced- mismatched parentheses — example: { (})

Algorithm to Check Balanced Parentheses

- The basic idea is to gather up open parentheses on the stack.
- When you find a close parenthesis, pop an element off the stack.
  - The current close parenthesis and the popped open parenthesis should match.
  - This is because the popped parenthesis is the most recent one we saw!
- When we are done, the stack should be empty.
  - All parentheses are matched.

```
while there are more tokens, read in a token
        if the token is an open parenthesis
            push the token onto the stack
        else (the token is a closed parenthesis)
            if the stack is empty
                the expression is unbalanced (we're done- return false because of extra closed parentheses)
            else (the stack is not empty)
                pop a token
                if the closed and open parenthesis don't match
                    the expression is unbalanced (we're done- return false because of mismatched parentheses)
// there are no more tokens left- the while loop is done
if the stack is empty
    the expression is balanced (we're done- return true)
else (the stack has tokens remaining in it)
    the expression is unbalanced (we're done- return false because of extra open parentheses)
```

Stack Example

- Review the example trace and code for balanced parentheses.

Queues

- First-in, first out
  - Last in, last out
  - Example: waiting in line
- Many uses in programming
  - Animation
  - Simulation
  - Networking
- Methods
  - enqueue / offer
  - dequeue / poll
  - getFront / peek
  - isEmpty

Practice

- Trace:

  offer(“Alice”)
  offer(“Bob”)
  peek
  poll
  offer("Dave")
  poll
  peek
- Write a method to simulate customers waiting in line. Allow the user to add new customers and wait on the next customer.

Other ADTs

- Priority Queue
  - Example: first class and coach passengers boarding a plane
- Set
  - Example: user names
- Map
  - Example: dictionary

GENERICS

Generics

- Generics allow you to specify a class when your data type is instantiated
- Example
  - ArrayList list = new ArrayList(); // can hold any type of object
  - ArrayList<Student> sList = new ArrayList<Student>(); // can hold only Student objects- this is enforced at compile time

Using Generics

- *Only* objects of one type (or a compatible type) can be added to the collection
  - Compatible type: child or descendent class!
- When an object is removed, Java already knows the type
  - You do not have to cast it!
- Puts compile-time checks on the types of objects in your collection.
  - Much better than testing at runtime!

Using Generics in Your Classes

- Using generics with the Java collections classes is straightforward and usually a good idea.
  - Most of the time, you want your collections to only hold one kind of object.
- You can also use Generics in your own classes.
  - To do this, you define a *type* for your class:

```
public class MyCollectionClass { …
```

becomes

```
public class MyCollectionClass<T> { …
```

Using Generics in Your Classes (cont.)

- Now, you can refer to the class T anywhere in your class.
  - T becomes a *placeholder* for any other class- String, Integer, Student, etc.
- When you create an object, you specify what T will be:

```
MyCollectionsClass<String> myStrings;
   // can only hold String objects
   // the class String will stand in or replace
     everywhere you put a T
MyCollectionsClass<Student> myClass;
   // can only hold Student objects
   // the class Student will stand in or replace
     everywhere you put a T
```

Practice

- Revisit the Course class and see what it looks like without generics.

Practice

- Write a class called Box that can hold a single item at a time.

Practice

- Review the Client/VIPClient example.
- Create a Roster that is a client list. Create methods to:
  - add or remove a client from the roster
  - determine if a client is on the roster
  - get the number of clients and number of VIP clients
  - generate a list of all VIP clients
  - calculate the total of all fees paid by all clients and all VIP clients
  - return a randomly selected client

Practice

- Write a class for a queue ADT.
  - Use a List behind-the-scenes.
  - Use generics.