Lab 4: Java Container Classes and Abstract Data Types

CS 320 Principles of Programming Languages, Fall Term 2019
Department of Computer Science, Portland State University

Learning Objectives

Upon successful completion, students will be able to:

- Write Java programs involving interfaces, implementations, and generics.
- Use Java library collection classes to build and apply simple data structures.

Instructions

Download the file `lab4.zip` from D2L into any convenient directory and type

```
$ unzip lab4.zip
```

You'll see a new directory `lab4` with a set of sub-directories, 00-11X. As usual, we'll walk through many or all of these directories, in sequence. You should try compiling and running the code in each directory as appropriate. In most directories, something like `javac *.java` will be appropriate for compiling. Directories whose names end in `X` contain exercises that you will need to turn in.

When you have completed all the exercises, return to the top-level directory above `lab4` and type

```
$ zip -r sol4.zip lab4/*X*/*
```

Then submit the resulting file `sol4zip` to D2L.

Note: These notes may be less complete (or simply less verbose!) than the ones for the previous labs. However, there are good on-line references to help with the material covered here. In particular, I recommend consulting

• Bruce Eckel, *Thinking in Java, 4th Ed.* available online at `https://archive.org/details/TIJ4CcR1`, especially the Chapter called "Holding Your Objects."

• The Java library API, especially for the `util` package, available at `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/package-summary.html`

Some of the code and data in this lab derives from materials on the website `https://algs4.cs.princeton.edu/`, copyright 2000-2017 by Robert Sedgewick and Kevin Wayne.

An Integer Sequence Interface (00)

We start with a simple `interface` for an abstract data type of integer sequences, i.e. a collection of `int` elements stored in a specific order. Our `Sequence` type is very similar to the Java `List` type, but it is significantly simpler, with only four operators.

"00/IntSequence.java"

```
interface IntSequence {

  // Append an element to the end of the sequence.
  void append(int e);

  // Prepend an element to the beginning of the sequence, shifting the
  // indices of all existing elements up by one.
  void prepend(int e);

  // Return number of elements in sequence
  int size();

  // Return the element at the specified position in the sequence.
  // Positions are numbered from 0.
  // throws IllegalArgumentException if the index is out of bounds
  int get(int index);
}
```

An `interface` file is the primary documentation for how the services provided by a library type are to be used. In addition to type signatures (which Java requires, of course), it should include comments about intended use, parameters, return values, and any exceptions that might be raised. However, the `interface` file is silent about how values of the type are to be represented and how the operations are to be implemented. Consequently, it usually doesn't say anything about performance characteristics of the various operations either.

Note also that the interface does not show constructors for the type, because these are tied to concrete implementations. Unfortunately, Java lacks a clean way to abstract over creation of ADT values.

One small point that can trip you up: all methods defined in an `interface` are *implicitly* declared to be `public`. But when defining a class that implements the interface, you must *explicitly* declare the corresponding method names to be `public`.

Integer Sequences via Linked Lists (01)

Next we look at one possible implementation of the `IntSequence` type, based on doubly-linked lists of nodes, each containing an integer value and pointers to each of its neighbors.

"01/ListIntSequence.java"

```
public class ListIntSequence implements IntSequence {
  private static class Node {
    int item;
    Node next;
    Node prev;

    Node(Node prev, int element, Node next) {
      this.item = element;
```

```
9           this.next = next;
10          this.prev = prev;
11        }
12      }
```

Nodes are represented using a nested private class `Node`. Objects of this class can only be created or accessed from within the enclosing `ListIntSequence` class. This is exactly what we want here, because the fact that we're using a linked list to store our sequence is an implementation decision that we want to hide from clients.

Each objects of the `ListIntSequence` type has three member fields

"01/ListIntSequence.java"

```
14    private Node first;
15    private Node last;
16    private int size;
```

pointing to the head and tail nodes of the list, and tracking the total number of nodes (just for convenience).

"01/ListIntSequence.java"

```
24    public void append(int e) {
25      final Node l = last;
26      final Node n = new Node(l,e,null);
27      last = n;
28      if (l == null)
29        first = n;
30      else
31        l.next = n;
32      size++;
33    }
```

The code for adding to the list is standard. For example, to append at the end, we create a new node, reset `last` to point to it, and also reset the old last node (if any) to point to it as well. As usual, we have to do a little extra work to handle the case where the old list was empty. The code for `prepend` is just the mirror image of `append`.

"01/ListIntSequence.java"

```
50    public int get(int index) {
51      if (index < 0 || index >= size)
52        throw new IllegalArgumentException("index " + index  + " out of range");
53      if (index < (size >> 1)) {
54        Node x = first;
55        for (int i = 0; i < index; i++)
56          x = x.next;
57        return x.item;
58      } else {
59        Node x = last;
60        for (int i = size - 1; i > index; i--)
61          x = x.prev;
62        return x.item;
63      }
64    }
```

Finally, the code for `get` just involves counting along the list until the desired index is found. This incorporates one efficiency trick: if the index requested is more than half-way through the list, we count from the end instead of from the beginning.

Although algorithm analysis is not the point of this course, it should not be hard to see that the `append` and `prepend` operations are quite fast: they take a constant amount of time no matter how big the sequence grows. However, `get` can be quite slow: in the worst case (accesses towards the middle of the sequence), it takes time proportional (half) the length of the sequence.

Integer Sequences via Arrays (02)

If we are using the `IntSequence` type in an application where additions to the list are rare, but lookups are common, a different implementation may be more efficient. In this directory, we add an alternative implementation of `IntSequence`, called `ArrayIntSequence`, that is based on arrays instead of linked lists.

"02/ArrayIntSequence.java"

```
4     private int[] contents;
5     private int size;
```

Each object of the `ArrayIntSequence` class carries an integer array `contents` and a current `size` indicating how much of `contents` is actually in use.

"02/ArrayIntSequence.java"

```
37    public int get(int index) {
38      if (index < 0 || index >= size)
39        throw new IllegalArgumentException("index " + index  + " out of range");
40      return contents[index];
41    }
42
```

Array elements within the range `0` to `size-1` correspond directly to the sequence values, so the `get` method becomes trivial.

"02/ArrayIntSequence.java"

```
14    private void ensureOneMore() {
15      if (size >= contents.length) {
16        int newCapacity = contents.length * 2;
17        contents = java.util.Arrays.copyOf(contents,newCapacity);
18      }
19    }
20
21    public void append(int e) {
22      ensureOneMore();
23      contents[size++] = e;
24    }
25
26    public void prepend(int e) {
27      ensureOneMore();
28      System.arraycopy(contents, 0, contents, 1, size);
29      contents[0] = e;
30      size++;
31    }
```

Appending is almost equally easy: we just put the new value at the first unused position in the array—as long as it has one. The initial array size is set to an arbitrary smallish size (here 10). If adding a new element would overflow the array, we use the `Arrays.copyOf` library utility to replace it with a fresh one of twice the size. (Doubling the size instead of growing it one element at a time might waste space, but it can save a huge amount of time if we keep inserting sequence elements.)

Prepending is much more expensive, because we must shift all the existing array elements to the right. Again, we use a library routine, `System.arraycopy`, for that task.

Overall, we expect this implementation to do lookups very quickly (in constant time, independent of sequence size), appends very quickly except when an array copy operation is needed (in constant *amortized* time, independent of sequence size), and prepends quite slowly (in time proportional to sequence size).

Choosing Alternative Implementations (03)

This directory adds a simple test driver `IntSequenceTest` that can be configured to use either of the two implementations of `IntSequence` that we have defined, according to whether the first command line argument is `list` or `array`. This is used to control which constructor we use to create a sequence. The remainder of the test driver is identical regardless of implementation choice: it performs a number of appends and prepends, according to the other command line arguments, and then prints out the sequence contents. Experiment with the test program and convince yourself that the client code works identically on both implementations.

Performance Testing the Alternatives (04X)

This directory has another test program `IntSequenceTimer`, a variation on `IntSequenceTest` intended to help gather some performance information about our two implementation alternatives. It takes five command line arguments:

- Which implementation to use (`list` or `array`)
- How many times to repeat the test.
- How many appends to perform.
- How many prepends to perform.
- How many reads (from the middle of the sequence) to perform.

Use this program to confirm or deny some of the claims made in earlier sections about the relative performance of the two implementations on different kinds of sequence operations. To do this, compare the computing times required for different choices of command line arguments, using the `time` shell command. For example, suppose we claim that reading from the middle of a sequence is much faster for `ArrayIntSequence` than for `ListIntSequence`. To confirm this, we might run the following exprerirment:

```
$ time java IntSequenceTimer list 10000 500 0 500

real 0m1.938s
user 0m1.884s
```

```
sys 0m0.065s
$ time java IntSequenceTimer array 10000 500 0 500

real 0m0.159s
user 0m0.132s
sys 0m0.051s
```

(Note: This is the output of the `bash` shell's `time`; other shells may format the output differently. For most accurate results, using the sum of the user+sys time is best. This probably doesn't work on Windows at all; you'll need to do this exercise on a linux or MacOS setup.) This times 10000 repetitions of a test in which we append 500 integers, prepend 0 integers, and read 500 integers from the middle of the sequence, varying the implementation. It shows that the `list` implementation is roughly 10 times slower than the array implementation.

Specifically, your task is to try to confirm (or deny!) the following claims:

1. For arrays, prepending is much slower than appending.
2. For lists, prepending and appending are roughly the same speed.
3. Prepending is much faster in the list implementation than in the array implementation.
4. Appending is roughly the same speed for both implementations.

For each of these, design an experiment with two different sets of parameter values. You are looking for large effects: "much faster/slower" means "slower/faster by a factor of 5 or 10X." "Roughly the same speed" means "within a factor of 2X". Careful: don't make the numbers too big or your program will run a long time and eat up all resources on the machine you're running on. On the other hand, don't make them so small that your `time` reports times of less than 0.1s. And remember that some effects are only visible when the sequence has 100s of elements. Aim for times of 0.1s to 5.0s and CTRL/C your program if things go much longer than that.

Your answer should go into a new file `timing.txt`, which is what we will use to grade this exercise. For each of the four claims, give:

- the two sets of parameter values you used;
- a written explanation of what you expect experiment to tell you;
- the actual timings obtained; and
- a written analysis based on the timings of whether the claim is true or not.

The written analyses should be brief: one or two sentences will do.

Generalizing the Interface: Subtype Polymorphism (05)

Suppose that instead of a sequence of `int` values we wanted to build a sequence of `float` values or `String` objects or `Foobar` objects. The logic for doing this is exactly the same as for `IntSequence`s but unfortunately, we cannot re-use our interface or implementations because they hardwire in the fact that elements are `int`s.

Code that works on values of more than one type is called *polymorphic*. (Accordingly, code that works on values of only one type is called *monomorphic*.) There are several approaches to rewriting our code to make it more more polymorphic. Perhaps the simplest thing is just to replace the element type `int` by `Object` in the signatures of `append`, `prepend`, and `get`, and in the representation arrays or nodes.

"05/ObjSequence.java"

```
8   interface ObjSequence {
9
10    // Append an element to the end of the sequence.
11    void append(Object e);
12
13    // Prepend an element to the beginning of the sequence, shifting the
14    // indices of all existing elements up by one.
15    void prepend(Object e);
16
17    // Return number of elements in sequence
18    int size();
19
20    // Return the element at the specified position in the sequence.
21    // Positions are numbered from 0.
22    // throws IllegalArgumentException if the index is out of bounds
23    Object get(int index);
24  }
```

"05/ArrayObjSequence.java"

```
4   private Object[] contents;
```

"05/ListObjSequence.java"

```
2   private static class Node {
3     Object item;
4     Node next;
5     Node prev;
6
```

Because every Java class is a subclass of `Object`, this interface allows objects of any kind to be stored into the sequence. This is called *subtype polymorphism*.

"05/ObjSequenceTest.java"

```
12  for (int i = 0; i < appends; i++)
13    seq.append("A");
14  for (int i = 0; i < prepends; i++)
15    seq.prepend(3.14);
```

The flexibility of this code is illustrated by the two calls at lines 13 and 15. The first shows that we can store a `String` object into the sequence. The second appears to store a `double` into the sequence. So not only can we use this code to build sequences of arbitrary type, we can store values of *different* types into the *same* sequence. Such collection objects are said to be *heterogeneous*, as opposed to collection objects containing ust a single type of element, which are called *homogeneous*.

Autoboxing and autounboxing. What is going on in line 15 is actually more subtle. While every object is indeed an instance of `Object`, `double`, like `int`,`long`,`boolean`, etc., is a *primitive type*, so its values are not objects. So it is not clear how they can be used where an `Object` is expected. The answer is that the Java library defines a set of *wrapper* classes, one for each primitive type, named `Integer`, `Float`, `Boolean`,

and so on. Each object of the `Double` class contains a single immutable `double` field (and similarly for the other wrapper classes). You can think of these objects as "boxes" containing a primitive value. While we cannot treat an `int` or `float` directly like an object, we can use an `Integer` or `Float` instead.

Moreover, Java *automatically* inserts a call to create a wrapper object whenever it can deduce that one is needed. This process is called *autoboxing*. That is, what is really compiled at line 15 is something like:

```
seq.prepend(Double.valueOf(3.14))
```

where `Double.valueOf(double d)` invokes a constructor for the wrapper class `Double` on parameter `d`. Similarly, Java also performs *autounboxing* to fetch the primitive value out of a wrapper object when it can deduce that this is needed. For example, the apparently type-incorrect code in the second line of

```
Double dobj;
double d = dobj;
```

will actually compile into

```
  double d = dobj.doubleValue()
```

where `double doubleValue()` is a method of class `Double` that returns the bare primitive value from the box.

Since almost all data ultimately gets stored in primitive values, autoboxing and autounboxing make it much easier to use polymorphism in Java.

Downcasting. There is one significant downside to subtype polymorphism. Most often, we use a container object of a class like `ObjSequence` to store values of one particular type (i.e. as a homogeneous container). If we only put, say, `String` values into the container, then surely anything we get back out of the container must be a `String`. But the types of the interface don't tell us this. Consider line 23 of `ObjSequenceTest.java`: it simply says that `get` returns an `Object`; it doesn't say what kind. If we want to use the result of a `get` operation, we must explicitly *downcast* it from type `Object` to the subtype that (we believe) it has, e.g.

```
ObjSequence seq;
String s = (String) (seq.get(0));
```

Such explicit downcasting is verbose and distracting. More importantly, it is dangerous: downcasting can fail at runtime if we are wrong in our assumption that the `Object` in question really has the subtype we expect. This kind of downcasting failure is a *checked* run-time error in Java (it raises an exception), so it doesn't destroy the type safety of the system. But it does open the possibility of unexpected aborts in released software, which is a bad thing.

Generalizing the Interface: Generics (06)

Fortunately, Java has a better solution to this problem, namely *generics*. Class, interface, and static method definitions can be given special *type parameters*, which stand for specific, but unknown, types to be supplied later. The full syntax is a little complex to master, but it is not hard to read examples.

"06/Sequence.java"

```
8  interface Sequence<E> {
9
10   // Append an element to the end of the sequence.
11   void append(E e);
12
13   // Prepend an element to the beginning of the sequence, shifting the
14   // indices of all existing elements up by one.
15   void prepend(E e);
16
17   // Return number of elements in sequence
18   int size();
19
20   // Return the element at the specified position in the sequence.
21   // Positions are numbered from 0.
22   // throws IllegalArgumentException if the index is out of bounds
23   E get(int index);
24 }
```

The <E> in line 8 parameterizes the interface by a type variable E. This variable is in scope throughout the interface definition, so can be used to specify argument types for append, prepend, and the return type of get.

"06/ListSequence.java"

```
1  public class ListSequence<E> implements Sequence<E> {
2    private static class Node<E> {
3      E item;
4      Node<E> next;
5      Node<E> prev;
6
7      Node(Node<E> prev, E element, Node<E> next) {
8        this.item = element;
9        this.next = next;
10       this.prev = prev;
11     }
12   }
13
14   private Node<E> first;
15   private Node<E> last;
```

Implementations are parameterized in a similar way. Note that the nested class Node must also be parameterized, so that it can hold items of the appropriate type.

"06/SequenceTest.java"

```
7    Sequence<String> seq = null;
8    if (implementation.equals("list"))
9      seq = new ListSequence<String>();
10   else if (implementation.equals("array"))
11     seq = new ArraySequence<String>();
12   for (int i = 0; i < appends; i++)
13     seq.append("Z");
14   for (int i = 0; i < prepends; i++)
15     seq.prepend("A");
16   int len = seq.size();
17   for (int i = 0; i < len; i++) {
```

```
18        String s = seq.get(i);
19        System.out.print(s + " ");
20      }
```

To construct an object of a parameterized class, we must provide the type parameter, as at lines 9 and 11 above. Here we are using `String`. The payoff comes in line 18, where we do not have to downcast the result of `seq.get()`; it is statically guaranteed to be a `String`.

In short, wherever we're tempted to use an `Object`, we should probably use a generic type variable instead. That's all there is to it.

Generics were not part of the original Java language, and they have a few nasty corners. Also, because there is still pre-generics Java code out in the world, the library (which uses generics heavily) still works with it, although it produces warnings. When you see a suggestion to compile with the `-Xlint:unchecked` flag, you should always follow it; it means you have made a mistake with generics that is tolerated by the compiler but may lead to runtime errors. Occasionally, it is impossible to avoid these warnings and it is best to turn them off using a *code annotation*; there is an example at line 8 of `ArraySequence.java`, due to one of the nasty corners: it is impossible to create an array of a generic type, so we must create an array of `Objects` and downcast it. Code like this is best left to the internals of libraries.

An Interface for Undirected Graphs (07)

Thus far, we have been working our own home-grown `Sequence` types. For the rest of the lab, we focus on using existing Java libraries, including `List` (the Java library equivalent of `Sequence`) but also `Set`, `Queue`, and `Map`. We will use these existing libraries to implement a *new* ADT that is not in the standard Java library, namely *undirected graphs*. (Note: there are plenty of graph libraries implemented in Java available on the web, but we are going to implement our own as a learning exercise.)

"07/Graph.java"

```
1   // Abstract Data Type for undirected graphs with vertices of type V
2   // Self-loops are allowed, but parallel edges are not
3
4   interface Graph<V> {
5
6     // Add a vertex. No-op if vertex already exists.
7     void addVertex(V v);
8
9     // Return all the vertices
10    Iterable<V> vertices();
11
12    // Return the number of vertices.
13    int vertexCount();
14
15    // Answer whether a particular vertex is in the graph
16    boolean hasVertex(V v);
17
18    // Add an edge between two vertices.
19    // Raises IllegalArgumentException if either vertex is not in graph
20    // No-op if edge already exists
21    void addEdge(V v1,V v2);
22
23    // Return the neighbors of a vertex
24    // Raises IllegalArgumentException if vertex is not in graph
```

```
25      Iterable<V> neighbors(V v);
26
27      // Return the degree (number of neighbors) of a vertex
28      // Raises IllegalArgumentException if vertex is not in graph
29      int degree(V v);
30    }
```

We start by giving an `interface` describing the operations we want to support on graphs. An undirected graph consists of a set of vertices and (undirected) edges between them. We will assume that graphs can have self-loops (there can be an edge from a vertex to itself) but not parallel edges (there is at most one edge between any two vertices). We want to be able to construct a graph dynamically, i.e. add vertices and edges as we go. (In a full ADT, one might also want to be able to delete vertices, extract subgraphs, etc., but we'll avoid these features to keep things simple.) The main questions we want to ask about a graph are: what are the vertices and what other vertices is each vertex directly connected to?

Here are some things to note about the interface:

- The `<V>` at line 4 specifies that the graph is generic over a type `V`, which is the type of vertex objects. Thus, for example, the `addVertex` method at line 7 takes a `V` as parameter. The client can instantiate `V` to any object type, including built-in library types like `String` or `Integer` (the `int` wrapper class), or a client-defined class.
- The `addEdge` method (line 21) requires that its two vertex parameters already be in the graph; if not, it rasises a standard Java library exception `IllegalArgumentException`. (A client can use `hasVertex()` to check first to avoid the risk of raising the exception.)
- The `vertices()` method at line 10 returns something that implements the interface type `Iterable<V>`. Basically, this means that the returned object supported iteration using the "foreach" construct, allowing the client to step through each vertex in the graph. Many standard library types, notably including `List` and `Set`, implement `Iterable`, so an implementation of `vertices()` can simply return one of these. The `neighbors()` method at line 24 is similar: it allows the client to iterate over all vertices that are directly connected to the parameter vertex.
- The `addVertex` and `addEdge` operations are designed to be *idempotent*, that is, adding a vertex or edge that already exists does not change the graph. That raises the question: when are two `V` values considered to be the same? The answer we choose is: when v1.equals(v2) returns true, where `equals` is a method defined on class `Object` and hence inherited by all objects. By default, `v1.equals(v2)` is defined to be the same as `v1 == v2`, i.e. it is true only if the two arguments are the *same* object (live at the same address). This is not very useful, so `equals` should be redefined for most classes, as it is for many standard library classes including `String` (where it means "has the same contents character by character") and the wrapper classes like `Integer` (where it means "wraps the same primitive `int` value). As long as we stick with classes like this, we should get intuitively reasonable behavior.

"07/GraphUtils.java"

```
4   public abstract class GraphUtils {
5     static <V> Graph<V> emptyGraph() {
6       return null; // just for compilation -- replace this with an actual constructor invocation
7     }
8
9     static <V> String dumpGraph(Graph<V> g) {
10      String s = "Vertex count = " + g.vertexCount() + "\n";
11      for (V v : g.vertices()) {
12        s += v + ":";
13        for (V w : g.neighbors(v))
```

```
14              s += " " + w;
15            s += "\n";
16          }
17          return s;
18        }
19
20    }
```

The auxiliary `GraphUtils` class contains some useful static methods for working on `Graph`s. `dumpGraph` illustrates the use of foreach to iterate through the vertices and their neighbors.

The `emptyGraph()` method is an example of a *factory* method. As noted, an `interface` never contains constructors; it is up to the client to pick a particular concrete implementation of the interface to use. It is often desirable to localize this choice in a single place separate from the main client code, so that it can be altered without recompiling the client. Factory methods wrap up the construction process; `emptyGraph()` just invokes a suitable constructor for some class implementing `Graph` and returns the result. For right now, we have no implementations, so to make the code in this subdirectory compile, we have `emptyGraph()` return `null` (which is a valid value of every class and interface, but of course won't work if we try to actually run the code).

"07/GraphTest.java"

```
1   public class GraphTest {
2     public static void main(String argv[]) throws Exception {
3       Graph<Integer> g = GraphUtils.emptyGraph();
4       g.addVertex(10);
5       g.addVertex(20);
6       g.addVertex(30);
7       g.addEdge(10,20);
8       g.addEdge(10,30);
9       g.addEdge(20,30);
10      System.out.println("Stage 1:");
11      System.out.print(GraphUtils.dumpGraph(g));
12      System.out.println("Graph contains vertex 10: " + g.hasVertex(10));
13      System.out.println("Graph contains vertex 50: " + g.hasVertex(50));
14      System.out.println("Degree of vertex 10 = " + g.degree(10));
15      System.out.println("Degree of vertex 30 = " + g.degree(30));
```

The `GraphTest` class provides a very simple test of graph construction and querying. At line 3, it invokes `GraphUtils.emptyGraph()` to get a graph satisfying the `Graph` interface instantiated at `Integer`. Note that the remaining client code neither knows nor cares which actual implementation of `Graph` is used. Of course, at the moment, this code cannot execute past line 4, because `g` is returned as `null`, but we'll fix that soon.

At line 4 and following, we take advantage of autoboxing to pass simple `int` values to `g.addVertex`; Java will automatically convert them to `Integer` objects.

Implementing the Graph Interface (08X)

This subdirectory contains the same files as in in 07/, with the addition of a skeleton for an *implementation* of the graph ADT, called `AdjSetGraph`. Your job is to complete this skeleton and test it using `GraphTest` and any additional tests you can think of. (Note that `GraphUtils.emptyGraph()` has been modified to invoke the `AdjSetGraph` constructor.)

The key implementation decision has been made for you: graphs are to be represented using adjacency sets. That is, for each vertex v, we explicitly represent its collection of neighbors $v_1, v_2, \ldots, v_n$. In more detail, a graph with vertices of type `V` is represented by a `Map` whose keys are vertices `V` and whose corresponding value for key v is a `Set` representing $\{v_1, v_2, \ldots, v_n\}$. We use a `Map` to support simple and reasonably efficient lookup of vertices no matter what type `V` is. We use a `Set` for the neighbors because the order of the vertices doesn't matter, and parallel edges are prohibited (so a given vertex can appear only once among the neighbors of another vertex).

One thing to note: the second type parameter to `Map`, namely `Set<V>`, is not just a simple type but rather a *type expression*. In general, types can be given by arbitrarily complicated expressions involving one or more layers of generic instantiation.

This exercise does not require writing much code, and that code is pretty simple. But it does require you to become acquainted with the details of the `Map` and `Set` libraries, using the web API documentation and whatever other sources you wish. In particular, you get to decide which implementations of `Map` and `Set` to use; read the documentation for advice.

Loading graphs from files (09)

This subdirectory is the same as 08, with these additions:

- `GraphUtils` contains a static method `readIntegerGraph` that reads a graph description file have file suffix `.ig` and constructs a `Graph` instance from it. Here's a typical example of an `.ig` file:

"09/tinyG.ig"

```
13
13
0 5
4 3
0 1
9 12
6 4
5 4
0 2
11 12
9 10
0 6
7 8
9 11
5 3
```

  Vertices are represented as small integers. The first line gives the number of vertices, the second line gives the number of edges; each remaining line describes an edge by giving the two vertices it connnects.

- `GraphUtils` also contains another static method `readStringGraph` that reads a graph description file have file suffix `.sg` and constructs a `Graph` instance from it. Here's a typical example of an `.sg` file:

"09/routes.sg"

```
JFK MCO
ORD DEN
ORD HOU
DFW PHX
JFK ATL
```

```
ORD DFW
ORD PHX
ATL HOU
DEN PHX
PHX LAX
JFK ORD
DEN LAS
DFW HOU
ORD ATL
LAS LAX
ATL MCO
HOU MCO
LAS PHX
STL PDX
```

Vertices are represented by strings. Each line of the file describes an edge by giving the two vertices it connects. In this file format, the separator between the two vertices is not necessarily as space (although it is in this example), so when we read a `.sg` file we need to supply the separator character. (By choosing a non-space character, we can allows the strings to contain spaces.)

- A classfile (only!) for a working implementation of `Graph` called `HiddenGraph` is provided, and `GraphUtils.emptyGraph()` has been patched to invoke its constructor. This allows you to compile and run the `GraphTest` program without having completed exercise 08X. (But if you have completed that exercise, you might want to invoke your implementation instead of the provided one.)
- `GraphTest` takes a file name on the command line invokes one or the other of the graph file readers and dumps out the results. Note: for an `.sg` file, you need to provide the separator character as an additional command line argument, e.g.

  ```
  java GraphTest routes.sg " "
  ```

Experiment a bit with the provided graph files, or write your own.

Dot output from graphs (10X)

This subdirectory is the same as the previous one, except that it a static method in `GraphUtils` to create a `.dot` file from a graph description. `dot` is a language for describing graph structures, which is an integral part of the AT&T GraphViz tools (see `http://www.graphviz.org`). In general, `dot` files can be used to produce output in a variety of graphical formats; we will use it to produce `pdf` files.

"10X/GraphUtils.java"

```
public static <V> void toDot(Graph <V> g, String gname) throws Exception {
  java.io.PrintStream out = new java.io.PrintStream(gname + ".dot");
  out.println("graph " + gname  + " {");
  for (V v : g.vertices())
    for (V w : g.neighbors(v))
      out.println("\"" + v + "\" -- \"" + w + "\"");
  out.println("}");
  out.close();
}
```

Running `GraphTest` now invokes this function instead of dumping a textual description of the graph. The file `foo.dot` can be converted to pdf and displayed by issuing the following command

```
dot -Tpdf -ofoo.pdf foo.dot
```

and then viewing `foo.pdf` in the pdf viewer of your choice.

The only problem is that the resulting displayed graphs have each edge represented twice! Modify the `toDot` function to fix this problem. Hint: this is easy if you use the right auxiliary data structure (a Java library class instance, naturally)!

Degrees of Separation (11X)

Finally, we'll use our `Graph` ADT (the `HiddenGraph` one or yours, whichever you like) to implement an algorithm to discover the *degree of seperation*, i.e. minimum length path, between any given vertices. This is a common metric in social networks, and is also the basis for the well-known Kevin Bacon game (google it!).

Your task is to write a program `DOS` that takes as command line inputs:

- a graph file name (`.ig` or `.sg`)
- (only if `.sg`) a seperator character
- a source node (integer or string as appropriate to file type)
- a target node (integer or string as appropriate to file type)

and produces as output the degree of separation (minimum path length) between the source and target, and *some* path (there may be more than one!) between them having that length.

For example, using the handy database of movie actors and the films in which they have appeared `movies.sg` (which has the separator `"/"`) we can play the Kevin Bacon game as follows:

```
$ java DOS movies.sg "/" "Bacon, Kevin" "Bogart, Humphrey"
Degrees of separation = 4
Sample path:
Bacon, Kevin
Novocaine (2001)
Martin, Steve (I)
Dead Men Don't Wear Plaid (1982)
Bogart, Humphrey
```

(Note that this database links actors to movies, not directly to each other, so degrees of separation reported here is 4 rather than the usual "Bacon number" of 2.)

The hard work for this task has already been done for you in the `BreadthFirstPaths` class, so study this closely before you begin! A breadth-first search starts at a given source vertex and visits all its immediate neighbors, then all its neighbors at distance 2, then at distance 3, and so on. Hence the path followed by the breadth-first search from the source vertex to a given target vertex is always of shortest possible length.