Topics: Vectors, Exception Handling

1 Vectors

- Vectors are one of the most important data structures in my opinion (along with hash tables and arrays). I use them any time I need a general expandable data structure.
- Vectors are expandable arrays; underlying storage structure is an array.
- Can be accessed using indices, like arrays.
- Elements can be added using addElement(); you don't have to explicitly set an index to a value.
- Vectors generally store `Objects`, so you can't use them to store primitives.
- A simple vector example:

```
public class SVector() { // A simple Vector class
  private Object[] store; // Can store any object.
  private int elements; // # of elements contained
  public SVector() {
    store = new Object[10]; // default size 10
    elements = 0;
  }
  // Assume for simplicity size >= current size.
  public void resize(int size) {
    Object[] temp = new Object[size];
    // Copy all elements into new array.
    System.arrayCopy(store, 0, temp, elements);
    store = temp;
  }
  public void addElement(Object o) {
    if (elements == store.size) {
      resize(2 * elements); // Double size of array.
    }
    store[elements] = o;
    elements++;
  }
  // Access elements using indices.
  public Object elementAt(int index) {
    return store[index];
  }
}
```

- The Java `Vector` class provides more functionality than our simple example. Read the online documentation.

2 Exception Handling

2.1 Method Call Stack

In any arbitrary program, the method calls are arranged as a *stack*. A stack is a data structure that only allows you to add something to the top or to remove something from the top, like a stack of books. It does

not permit adding something in or removing something from the middle of the stack. Let's use the following rules in adding and removing a method from our stack:

1. We add a method to the stack when it is called.
2. We remove a method from the stack when it returns.

These rules result in a few invariants that hold at all times in a program:

1. The method at the top of the stack is the method that the program is executing.
2. A method only returns when it is at the top of the stack.

It is easy to see why these invariants hold. The first is true because a method is placed on the stack when it is called, and a method must have been the last one called (without returning) in order to be the one executing. The second must hold for rule #2 to work.

2.2 Exception Handler Stack

Similar to the method call stack, there also exists an exception handler stack in each program. For this stack, an exception handler (a `catch()` block) is added when its corresponding `try` block is entered and removed when the `try` block is left. When an exception occurs, the stack is searched from top to bottom, and the first matching exception handler is called (the first `catch()` block for which the declared type matches). Consider the following program as an example:

```
static void main(String[] args) {
  try {
    ... // some code here
    foo();
    ... // some code here
  } catch (IOException e) {
    ... // exception handler A
  }
}

static void foo() throws IOException {
  try {
    ... // some code here
  } catch (IOException e) {
    ... // exception handler B
  }
  bar();
}

static void bar() throws IOException {
  ... // some code here
  if (...) {
    throw new IOException();
  } else {
    ... // some code here
  }
}
```

Say in one execution of the program, an exception is thrown in the method `bar()`. When the first `try` block in `main()` was entered, exception handler A was added to the exception handler stack. Then `foo()` was called, and its `try` block entered, so exception handler B was added to the stack. However, the `try` block was then exited normally, so B was removed from the stack. Then `bar()` was called. So when the exception

is thrown, the only handler on the stack matchin `IOException` is A, so the program jumps to that `catch()` block.

On the other hand, consider a slightly different program:

```
static void main(String[] args) {
  try {
    ... // some code here
    foo();
    ... // some code here
  } catch (IOException e) {
    ... // exception handler A
  }
}

static void foo() throws IOException {
  try {
    ... // some code here
    bar();
  } catch (IOException e) {
    ... // exception handler B
  }
}

static void bar() throws IOException {
  ... // some code here
  if (...) {
    throw new IOException();
  } else {
    ... // some code here
  }
}
```

Now the call to `bar()` occurs inside the `try` block in `foo()`. So at this point, exception handler B is still on the stack and is at the top since it was added last. So when the exception occurs, B will be executed since it is the topmost exception handler that matches `IOException`.

Again, let's modify the program slightly:

```
static void main(String[] args) {
  try {
    ... // some code here
    foo();
    ... // some code here
  } catch (IOException e) {
    ... // exception handler A
  }
}

static void foo() throws IOException {
  try {
    ... // some code here
    bar();
  } catch (ArrayIndexOutOfBoundsException e) {
    ... // exception handler B
  }
}
```

```
static void bar() throws IOException {
  ... // some code here
  if (...) {
    throw new IOException();
  } else {
    ... // some code here
  }
}
```

The exception handler stack looks exactly the same as the previous program when the exception occurs, except that B's type does not match `IOException` anymore. Since the topmost handler that matches `IOException` is A, it will be the one that is executed. There is a subtle point here though. If the type of B was `Exception` instead of `NullPointerException`, B would be the one executed. This is because `IOException` is a subtype of `Exception`, so the types will match.

More complicated examples where a `try` block has more than one associated `catch()` block can be constructed. In such a case, the associated `catch()` blocks are added to the stack in *reverse* order from which they appear in the code. The rules for choosing which handler is used are then the same as above.