CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #7

John Ridgway

February 19, 2015

1 Generic Collections

Collection Elements

- A stack is an example of a *collection*. Most common data structures are collections.
- A collection usually holds a group of data items of some common type; e.g., `Locomotive`, `Dog`, `String`, or `int`.
- Consider `StringLog` which is a named collecton of `String`s. If we wanted a class `IntegerLog`, we could simply copy `StringLog` and then replace every use of `String` with a use of `Integer`.
- At a different level this is exactly what we don't want you to do with methods — no cutting and pasting.

Collection Elements (continued)

- Alternatively, we could define one class `Log` that holds `Object`s. Pro: we can put any kind of element in such a log. Con: once we've extracted an item we have to *cast* it to the appropriate type before we can use.
- Better, we can use *generics* (introduced in Java 5) to write a single `Log` that can be parameterized to hold elements of a particular type.
- We will be using *generics* throughout the rest of the course.

Generic Interfaces and Classes

Imagine creating a new class `Pair` to return two values from a single method, as shown here:

```
class Pair {
  private String l;
  private Dog r;
  Pair(String l,
       Dog r) {
    this.l = l;
    this.r = r;
  }
  String getL() {
    return l; }
  Dog getR() {
    return r; } }
```

```
Pair gen() {
  ...
  return new Pair("", d);
}
...
void user() {
  Pair result = gen();
  result.getL();
  result.getR();
  ...
}
...
```

Generic Interfaces and Classes (continued)

Now we have another place where we want to return two values, one of class `S` and one of class `T` from a method. Copy the `Pair` class? No, make it generic:

```
class Pair<L, R> {
  private L l;
  private R r;
  Pair(L l, R r) {
    this.l = l;
    this.r = r;
  }
  L getL() {
    return l; }
  R getR() {
    return r; } }
```

```
Pair<S,T> gen() {
  ...
  return new Pair<S,T>(s,t);
}
...
void user() {
  Pair<S, T> result = gen();
  result.getL(); // An S
  result.getR(); // A T
  ...
}
...
```

Generic Interfaces and Classes (continued)

- When you create a generic interface or class you provide a type variable, or multiple type variables, inside `<` and `>` immediately following the class or interface name.
- When you want to use one you specify actual types (must be a class, not a primitive type).
- Just like formal and actual parameters, but “lifted” a level.

Exceptional Situations

- Suppose a piece of code runs into an exceptional situation. What is it to do?

- Prior to exceptions the code had to return a special value to indicate that there was a problem. (Consider `fopen` in C.)
- Programmers are lazy, and frequently fail to check for those special values.
- Exceptions were invented to mitigate this problem. *Throwing* an exception halts the normal flow of control and transfers control somewhere special. By default Java prints a message and halts the program.

Exceptions

- A piece of code can throw any instance of `Throwable`. It should always be an instance of `Exception` or `Error`. Any exception that you create or throw should be an instance of `Exception` (or a sub-class thereof).
- `throw new Exception("explanation");`
- `Error` is for system problems; there's nothing you can do about them.
- `RuntimeException` is also for things you are unlikely to be able to do anything about, usually programming errors. `NullPointerException` is a subclass of `RuntimeException`.

Checked Exceptions

- Anything that can be thrown, except `Error` and `RuntimeException` is considered to be a *checked* exception, i.e., it must either be caught in the method that throws it, or the declared in its `throws` clause.
- Similarly, if a method calls another method that declares that it can throw a particular exception, then the caller must either catch the exception or declare that it, in turn, can throw that exception.
- This means that the caller of a method knows exactly what exceptions can be thrown by anything it calls.

Exception Example

```
Reader getFileToRead(BufferedReader input)
  throws IOException
{
  Reader r = null;
  while (r == null) {
    System.out.println("Enter file name: ");
    String fileName = input.readLine();
    try {
      r = new FileReader(fileName);
    } catch (FileNotFoundException e) {
      System.out.println("File not found.");
    }
  }
  return r;
}
```

Programming By Contract

- Programming by contract is the practice of writing methods that don't deal with the cases where their preconditions are false.
- If it's necessary to test some precondition, then it shouldn't be a precondition.
- If I have a method `getInt` that needs to get an `int` from the console, the code that tells the user "Not an int, try again" should be within this method.
- Any other method should be able to assume that `getInt` will return an `int`.

The Two Stack Interfaces

- DJW define three interfaces in order to have both bounded and unbounded stacks.
- `BSI<T>` and `USI<T>` differ in that only `BSI<T>` has an `isFull` method, and they have different `throws` clauses. (Since these exceptions are not checked, these `throws` clauses are actually just advice for the programmer.)
- Note that DJW's `pop` does not return the element popped — you have to get it with `top` first if you want to save it.

The Stack Interfaces

```
public interface SI<T> {
  void pop( )
    throws StackUnderflowException;
  T top( )
    throws StackUnderflowException;
  boolean isEmpty(); }

public interface BSI<T> extends SI<T> {
  void push(T element)
    throws StackOverflowException;
  boolean isFull( ); }

public interface USI<T> extends SI<T> {
  void push(T element); }
```

Stack Interfaces Clicker Question

Consider the three interfaces we have just seen. Suppose I write a generic class `ArrayStack<T>` that implements the interface `BSI<T>`. Which of the methods `pop`, `push`, `top`, `isEmpty`, and `isFull` must be implemented in `ArrayStack<T>`?

A. `push`, `top`, and `pop` only

B. all of them

C. all but `isFull`

D. only `push` and `isFull`

2 Stacks: Array Implementation

Stacks: Array Implementation

- Data fields and constructors
- Transformers: pushing and Popping
- Observers: `isEmpty`, `isFull`, `getSize`, `peek`

ArrayStack: Data Fields and Constructors

```
public class ArrayStack<T> {
  private int currentSize = 0;
  private T[] contents;

  public ArrayStack(int size) { // TROUBLE
    contents = (T[])(new Object[size]);
  }
```

ArrayStack: Pushing and Popping

```
public void push(T item)
    throws StackOverflowException {
  if (isFull()) {
    throw new StackOverflowException(); }
  contents[currentSize] = item;
  currentSize += 1; }

public T pop()
    throws StackUnderflowException {
  if (isEmpty()) {
    throw new StackUnderflowException(); }
  T top = contents[currentSize];
  contents[currentSize] = null;
  currentSize -= 1;
  return top; }
```

ArrayStack: Observers

```
public int getSize() {
  return currentSize; }

public boolean isEmpty() {
  return currentSize == 0; }

public boolean isFull() {
```

```
    return currentSize >= contents.length; }

  public T peek()
      throws StackUnderflowException {
    if (isEmpty()) {
      throw new StackUnderflowException(); }
    return contents[currentSize - 1]; } }
```

ArrayStack Question 1

Which of these statements will **not** remain true as the stack operates normally?

A. the number of stack elements stored in the array is `size`.

B. the top element of the stack is stored in location `size`.

C. If `x >= size`, location `x` of the array is `null`.

D. If the stack is not full, the last location is `null`.

ArrayStack Question 2

```
  public void push(T item)
      throws StackOverflowException {
    if (isFull()) {
      throw new StackOverflowException(); }
    contents[currentSize] = item;
    currentSize += 1; }
```

What would happen if we reversed the two statements after the `if` statement?

A. The code would not compile.

B. A subsequent `pop` operation would return the wrong thing.

C. The stack would be in reverse order.

D. There would be an exception the first time we pushed an element onto an empty stack.