# COE318 Lecture Notes Week 6 (Week of October 6, 2014)

## *Announcements*

- Midterm (20% of total mark) on Wednesday, October 15. Covers weeks 1–6, Labs 1–5.
- My counselling hours:
  - Tuesday 2—3pm
  - Wednesday 3—5pm
  - Friday 1—2pm
- I am in the process of re-reading (and fixing) all my lecture notes. All source code examples will have links so you can easily download the Java code or Netbeans project. (The revision of the notes will be completed by next Tuesday morning at the latest.)
- Previous midterms (with answers) are available on the Web site.
- More questions will be added to the Study Guide daily.

## *Topics*

- Java is "pass by value"
- Casting primitives
- Converting Strings to numbers
- APIs
- ArrayList
- Notes on Lab 5
- Design and implementation: an "in class" interactive demonstration
- Q&A

## *Java is "pass by value"*

- When a method is invoked, the parameters passed are ***copies***. Thus if the method modifies a parameter, its modified value is invisible to the caller.
- However, when a reference variable is passed, the original parameter (before being copied) cannot be changed **BUT** the object it references can be changed by the method.
- (This is one reason why *immutable objects* are considered safer.)
- Consider the following for example:

```
package coe318;

public class Goo {

    private int i;

    public Goo(int i) {
        this.i = i;
    }

    public void f(int j, int k, Goo g) {
        j++;
        k--;
        g.i = k;
    }

    public static void main(String[] args) {
        int j = 5;
        int k = 6;
        Goo h = new Goo(2);
        System.out.println("Before h.f(...) j: " + j + ", k: "
                + k + ", h.i:" + h.i);
        h.f(j, k, h);
        System.out.println("After h.f(...) j: " + j + ", k: "
                + k + ", h.i:" + h.i);
    }
}
```

The output when this is run is:

```
Before h.f(...) j: 5, k: 6, h.i:2
After h.f(...) j: 5, k: 6, h.i:5
```

## *Casting primitives*

- Primitive data types (`byte, char, boolean, short, int, long, float, double`) have different sizes and ranges.
- The table below summarizes the characteristics of the primitive data types.
- The boolean data type cannot be assigned to or from any other data type (unlike C).
- When a numerical data type is assigned to another numerical type that is smaller (either in size or range), the compiler insists that a cast must be used.
- For example:

```
int zero = 0.0;  //ILLEGAL must cast floats to ints
int zero = (int) 0.0;  //OK, but silly
int zero = 0;     //Better
```

- When a larger size is assigned to a smaller size type, it is *truncated* (not rounded). For example:

```
double x = 5.99999999999;
int i = (int) y; //i will have value 5, cast necessary
char ch = 0xffff;
short s = (short) ch; //cast required, s has value -1
int i = ch; //no cast needed, i has value 65535
```

| Type | Size | Range |
|---|---|---|
| `byte` | 1 byte (8 bits) | -128...+127 |
| `boolean` | 1 byte (8 bits) | ***true*** or ***false*** |
| `short` | 2 bytes (16 bits) | $-2^{15}...2^{15}-1 (-32768... + 32767)$ |
| `char` | 2 bytes (16 bits) | $0 \dots 2^{16}-1$ |
| `int` | 4 bytes (32 bits) | $-2^{31} \dots 2^{31}-1$ |
| `long` | 8 bytes (64 bits) | $-2^{63} \dots 2^{63}-1$ |
| `float` | 4 bytes (32 bits) | $\pm 3.4 \times 10^{-38} \dots 3.4 \times 10^{38}$ |
| `double` | 8 bytes (64 bits) | $\pm 1.7 \times 10^{-308} \dots 1.7 \times 10^{308}$ |

*Table 1: Java Primitive Data Types (size and range)*

## Converting a String to a numerical data type

- Corresponding to each primitive data type, there is a class that encapsulates it. These classes are:
  - `Boolean`
  - `Byte`
  - `Character`
  - `Long`
  - `Integer`
  - `Float`
  - `Double`

- They all contain static methods such as `parseDouble` (in the Double class) or `parseInt` (in the Integer class).
- For example:

```
String strs = {"12", "23", "5", "6"};
int total = 0;
for(String s : strs) {
      total += Integer.parseInt(s);
}
```

## Application Programming Interface (API)

- Classes come with documentation describing how to use them. This documentation is called the class's *API*.
- In Java, the API is incorporated into the source code itself of the class through ***javadoc*** comments.
- These comments begin with `/**` and end with `*/`. They describe the public method, constructor or variable whose declaration follows. (Anything public or protected should be documented. Private members can optionally be documented as well.)
- A tool then converts the javadocs into html pages.
- The API constitutes ***the design*** of the class.
- The API (javadocs) should be written *before* implementation.
- The API should be sufficient for a programmer to know what to implement and for a tester to write test code for the implementation. (In other words, it is common to write test code even before the actual code exists!)

## ArrayList class

### Arrays are Fixed Size—Making them appear to be changeable

- Fixed size array can be annoying. If the size of the array is too small, the program will crash. If it is too big, memory is wasted.
- The class below—`GrowableArray`–shows how you can create a class that "behaves" like a "resizable array" of Strings.
- The "trick" is to store the Strings in a **private** array of fixed size. When the array is full and another String is to be placed in the `GrowableArray`, a new String array double the size is created and the elements of the old array are copied to it. Finally, the private instance variable containing the String array reference is set to the new larger array. (The old array is then

eligible for garbage collection.)

- The code follows (and note that javadoc comments are included that can generate the API documentation.)

```
package coe318;

public class GrowableArray {

    /**
     * The <code>GrowableArray</code> class represents a
collection of any
     * number of <code>String</code>s. Unlike an array with fixed
size
     * determined at compile time, a <code>GrowableArray</code>
object will grow
     * to be as big as necessary.
     */
    private String[] strings;
    private int nSlotsUsed;

    /**
     * No-arg constructor
     */
    public GrowableArray() {
        this.nSlotsUsed = 0;
        strings = new String[2];
    }

    /**
     * Adds a String to the  <code>GrowableArray</code>.
     *
     * @param s The String to add.
     */
    public void add(String s) {
        if (nSlotsUsed == strings.length) {
            System.out.println("Making array BIGGER");
            String[] biggerArray = new String[2 * strings.length];
            for (int i = 0; i < strings.length; i++) {
                biggerArray[i] = strings[i];
            }
            strings = biggerArray;
        }
        strings[nSlotsUsed++] = s;
        System.out.println("        Added " + s);
    }
```

```
    /**
     * Returns the i'th added String.
     *
     * @param i The position of the String.
     * @return The String at position "i".
     */
    public String get(int i) {
        return strings[i];
    }

    /**
     *
     * @return The number of Strings in the collection.
     */
    public int getSize() {
        return nSlotsUsed;
    }

    /**
     * A demonstration of a <code>GrowableArray</code> object.
     *
     * @param args Command line args (not used here)
     */
    public static void main(String[] args) {
        GrowableArray s = new GrowableArray();
        s.add("abc");
        s.add("xyz");
        s.add("AB");
        s.add("DE");
        s.add("WX");
        for (int i = 0; i < s.getSize(); i++) {
            System.out.println(s.get(i));
        }
    }
}
```

## *Using the standard ArrayList class*

- The `GrowableArray` class can only be used for Strings. A built-in class `ArrayList` can be used for any kind of object and has many more capabilities.
- The API documentation is available here.
- You need to use this class in Lab 5 (the Blackjack lab); for example in the `CardPile` class.

- The most common ways to use an `ArrayList` object are:

```
ArrayList<Person> people; //Declare an ArrayList of Person objects
people = new ArrayList<>(); //Instantiate it with constructor
Person p1 = new Person("Alice");
people.add(p1);     //Add Alice to the people ArrayList
people.add(new Person("Bob")); //Add Bob
//Print all people
for(Person p : people)
      System.out.println(p);
```

## *References (text book and online)*

- *Head First Java*: Chapter 6