Arizona State University

School of Computing, Informatics, and Decision Systems Engineering

CSE 205 Lecture Notes

Ryan Dougherty
redoughe@asu.edu

To my friends and family

Table of Contents

Introduction

Welcome to these lecture notes for CSE205! This guide will follow up on the first lecture notes (for CSE110), and get more into the "algorithms and computation" side of programming. However, all examples will be in Java, to remain consistent between the two sets of notes. As we have said before, programming is becoming more of an important tool in the last few decades. Understanding of these technological shifts, therefore, is imperative.

The topics are split up according to the Table of Contents above, and their corresponding page numbers. At the end of each chapter are written and programming exercises. The written exercises are for testing your knowledge of the material, and the programming ones are to see if you can write programs using the material from that chapter.

If there are any questions/errors/comments that you want to send regarding the material, send an email to: Ryan.Dougherty [at] asu.edu.

Useful Bits & Pieces

In this section we will give a few "miscellaneous" bits of information that are not big enough topics to warrant a whole section, but are nevertheless important. Knowledge from the first set of lecture notes is assumed.

2.1 String.split

There is a useful method in the `String` class called `split(String)` - it returns a `String` array (`String[]`) that consists of the `String` being called on without all instances of the passed in `String`. The passed in `String` is often called the "delimiter." Here we give some examples:

```
1 String original = "I am going to learn Java!";
2 String[] allWords = original.split(" ");
3 // allWords = {"I", "am", "going", "to", "learn", Java!"};
4 String[] words2 = original.split("a");
5 // allWords = {"I ", "m going to le", "rn J", "v", "!"};
```

2.2 ArrayList

Dealing with ordinary arrays in Java (or any programming language) is quite a hassle. Therefore, there is a class in the `java.util` package called `ArrayList`, which is an object wrapper for an array. It is a *templated* type, which means it has angle brackets to denote what type is allowed in the `ArrayList` (this type must be an `Object`, not a primitive type). Here are some examples of its use:

```
1  ArrayList<Integer> ints = new ArrayList<Integer>(); // example
     of initialization - must be Integer, not int
2  ints.add(0);
3  ints.add(2);
4  ints.add(0); // can add 0 again
5  ints.add(1, 2); // add 2 at index 1
6  ints.remove(3); // remove element at index 3
7  int value = ints.get(1); // gets element at index 1
8  ints.set(0, 5); // sets index 0 to 5
9  if (ints.isEmpty()) {
10       System.out.println("The array is empty!");
11 }
12 // for loop over the array
13 for (int i=0; i<ints.size(); i++) {
```

```
14        System.out.println(ints.get(i));
15    }
16    ints.clear(); // ints has no elements
```

There is another class called `Vector`, also in the same package, that provides the same interface as `ArrayList`, but is thread-safe, whereas `ArrayList` is not. This means `Vector` is useful in multithreaded environments, as we will see in a later chapter.

Interfaces, Abstraction, & Polymorphism

In this section we will start the "real" part of this second set of lecture notes. This particular section deals with some object-oriented design principles, and Java supports them very well.

3.1 Interfaces

An interface has the same structure as a class, but instead has the `interface` keyword where `class` would go, and only consists of two things: method declarations (the signature of the method plus a semicolon), and constants (with the `final` keyword). An interface is a "contract" of sorts, where some normal class will "implement" (via the `implements` keyword). By doing this, that particular class must implement every single method listed in the interface - otherwise, it is a compiler error. For example, if we created an interface for a `Ship`:

```
1 public interface ShipInterface {
2       public void shoot(); // just a declaration, no
           implementation
3 }
```

We can then implement this interface in any class we see fit:

```
1 public class Ship implements ShipInterface {
2       // ...
3       // not including shoot() will have a compiler error
4       public void shoot() {
5             System.out.println("This ship did shoot!");
6       }
7       // ...
8 }
```

We can also implement multiple interfaces at once as comma-separated values:

```
1 public class A implements Interface1, Interface2, Interface3 {
2       // ...
3 }
```

3.2 Inheritance

Inheritance is a well-known technique in object-oriented programming to organize a lot of similar code in a hierarchical way. In this system, there are two kinds of classes: *children*

(or *base*), and *parent* (or *super*) classes. The way it works is that the child classes inherit some information from the parent class. For Java, a parent can have one or more children, but a child can have at most one parent. Also, one may only inherit from a class if it is not declared as `final`.

A child class that inherits from a parent class gets all of the `public` methods, but none of the `private` methods (it does get `private` instance variables, however). So how do we enforce encapsulation for child classes then? This is where the `protected` keyword comes in. It has the same behavior as `private`, but any classes that inherit from it can use it. It is somewhat more "free" than `private`, but also has encapsulation.

In Java, in order to inherit from a class, one must use the `extends` keyword:

```
1  public class A {
2      // ...
3  }
4  public class B extends A {
5      // ...
6  }
```

A way that inheritance is useful is for allowing code reuse for child classes. For example, one may use the `super` keyword to denote the parent method/constructor to call, and it calls the appropriate method/constructor in the parent class:

```
1  public class A {
2      public A() {
3          // ...
4      }
5  }
6  public class B extends A {
7      public B() {
8          super(); // calls A();
9          // ...
10     }
11 }
```

Also, for all of the instance variables declared in the parent class, one does not need to re-declare them in child classes. However, one must use the `this` modifier to do so:

```
1  public class A {
2      private int a;
3      public A() {
4          a = 0;
5      }
6  }
7  public class B extends A {
8      public B() {
```

```
9               this.a = 1; // must use this modifier
10      }
11  }
```

3.3 Abstract

Another keyword in Java is the `abstract` keyword. It is used for a class (that now has the `abstract` modifier) that says that "this method will be implemented in child classes, but I won't personally implement it." The `abstract` method in the parent class is just a method header (like would be in an interface, but now has the `abstract` modifier), and is a normal method in child classes. Here is an example:

```
1  public abstract class Person {
2       public abstract double computePay();
3  }
4  public class CEO extends Person { // no abstract
5       public double computePay() { // no abstract
6            return 100000.0;
7       }
8  }
```

3.4 Polymorphism

3.5 Written Exercises

1. **What is the meaning of declaring a class `abstract` and `final`? What do you think will happen?**