# 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

Now we get to the most important part of this section - polymorphism. It is a way, at runtime, for deciding which method to call when there is a hierarchy of classes, and is relevant to inheritance. For example, if we have the classes:

```
1 public class A {
2       public void m() { ... }
3 }
4 public class B extends A {
5       public void m() { ... }
6 }
7 public class C extends A {
8       public void m() { ... }
9 }
```

We can develop code that has variables that can be one or more of these different classes without having to rewrite code:

```
1 A a1 = new A(); // works
2 B b1 = new B(); // works
3 C c1 = new C(); // works
4 A a2 = new B(); // ok, B has all attributes of A
5 // B b2 = new A(); // not ok, A does not have attributes of B
6 a2.m(); // calls B.m() even though a2 has declared type A
7 a1.m(); // calls A.m() because a1 is of type A
```

## 3.5   Written Exercises

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

# Big-O Notation

In this short section we will introduce the idea of Big-O notation. It is a measure for analyzing asymptotic (i.e. as the input size "becomes large") behavior and time to run the algorithm.

## 4.1  Big-O

One concept that is often used in Computer Science is asymptotic notation, and Big-O notation is one of those. It is usually written of the form:

$$O(f(n))$$

where $f(n)$ is some function that depends on $n$. We will try to avoid the formal mathematics behind Big-O notation, but it essentially involves looking at an algorithm, and observing the behavior of it if the input is of size $n$. For example, if we were linearly searching through an array, on average, we will look at around $\frac{n}{2}$ elements - therefore, that will be the number of operations, on average, that we will do to run the algorithm. In Big-O notation, however, we discard any constant factors (or lower-order terms). Therefore, we say linear search runs in $O(n)$ time.

## 4.2  Example

Let's analyze the running time of Binary Search for Big-O notation. Each time we run through the loop, we effectively remove half of the search space. So, if the array size was $n$, we will perform $x$ steps, where $2^x = n$. Solving this equation for $x$ yields that $x$ is approximately $log_2 n$. Since $log_2 n = \frac{log(n)}{log(2)}$, binary search runs in $O(log(n))$ time.

## 4.3  Note

A few things to consider:

– Any operation that takes constant time is marked as $O(1)$.
– Any two operations with time $O(f(n))$ and $O(g(n))$ will, put together, have running time $O(f(n) + g(n))$. For example, $O(n) + O(n)$ corresponds to an algorithm that runs in time $O(n)$.

## 4.4  Best, Average, Worst Case

You may have noticed in trying to analyze other algorithms that there are multiple paths to take sometimes. How do you analyze this? You can in three ways: best, average, and worst cases. The best case corresponds to the absolute lowest amount of asymptotic time that is possible to complete the algorithm, and worst corresponds to highest. Average case is harder to study (how does one define "average"?), so we will leave this out. Most algorithm analysis does worst case, because it provides a guarantee on the running time - the algorithm is guaranteed to asymptotically run in less time than in the worst case.

## 4.5 Written Exercises

**1. What are the running times of {Selection, Insertion} Sort?**

**2. What is the Big-O notation of the function $f(n) = 3n^3 + 2n^2 + n$?**

# Linked List, Stack, Queue, Heap

In this section we will introduce 4 new data structures.

## 5.1 Linked List

A Linked List (`java.util.LinkedList`) is a very simple data structure. It consists of `Node`s that have a *data* member and a *pointer* which points to another `Node` (a "doubly linked list" has 2 pointers - i.e. one points "forward" and the other "backward"). This construction may be very similar to that of an array, but has distinct advantages:

– If one is at a particular `Node`, one can remove/add a `Node` in $O(1)$ time. This is not possible for arrays, which takes $O(n)$ time (shifting elements forward or back depending on the operation).

– The linked list does not need to be reallocated, since the pointer points to the next `Node`, which may or may not be near the current one in memory.

However, there are disadvantages:

– If one loses the beginning (sometimes called *head*) pointer, then there is no other way of accessing other elements in the linked list, since that beginning pointer was the only `Node` pointing to the second `Node`, which points to the third, etc.

– One needs to traverse the linked list to get to a specific index, since one cannot just access any given index - therefore, accessing a specific element takes $O(n)$ time (to traverse); for arrays, since we can index directly, this takes $O(1)$ time.

Despite these disadvantages, we will see two data structures - Stack and Heap - that can be implemented using a linked list in an efficient way.

## 5.2 Stack

A Stack (`java.util.Stack`) is a data structure that resembles a stack of books - you can insert books one at a time on the top of the stack, but you can only remove the top item from the stack one at a time. One cannot directly any element in the stack other than the top element (if it exists). A stack, therefore, is called a LIFO (last-in-first-out) data structure. The operations are as follows: `pop`, remove the top element and return it; `push`, add an element to the top of the stack; `peek`, see the top element without doing anything to it. All stack operations require $O(1)$ time, since they do not depend on the size of the stack.

A Stack is implementable using a Linked List, where the head pointer is the top of the Stack. When one adds an element to the top of the Stack, the head pointer changes to the new element, and the new element's pointer points to the original top element.

## 5.3  Queue

A Queue (`java.util.Queue`) is a data structure that resembles a line queue of people - people can be serviced from the front of the queue one at a time in order, and people can enter the back of the queue one at a time. A queue, therefore, is called a LILO or FIFO (last-in-last-out or first-in-first-out) data structure. The operations are as follows: `enqueue`, insert an element into the back of the queue; `dequeue`, remove the element at the front of the queue. All queue operations require $O(1)$ time, since they do not depend on the size of the queue.

A Queue is implementable using a Linked List, where one has a pointer to the front of the queue, and another to the back of the queue. It is much easier to implement a Queue using a doubly linked list than a singly linked list, but it is still possible.

## 5.4  Heap

## 5.5  Written Exercises